

AD-A258 901



①

AFIT/GCS/ENG/92-04

Formalization and Transformation of Informal Analysis Models  
Into Executable REFINE<sup>TM</sup> Specifications

THESIS

Mary M. Boom  
Captain, USAF

Bradley D. Mallare  
Captain, USAF

AFIT/GCS/ENG/92-04

612  
93-00062  
372

DTIC  
ELECTE  
JAN 07 1993  
S B D

Approved for public release; distribution unlimited

93 1 04 165

# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 1992	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE FORMALIZATION AND TRANSFORMATION OF INFORMAL ANALYSIS MODELS INTO EXECUTABLE REFINE <sup>TM</sup> SPECIFICATIONS			5. FUNDING NUMBERS	
6. AUTHOR(S) Mary M. Boom, Capt, USAF Bradley D. Mallare, Capt, USAF				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/92D-04	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Douglas White RL/C3CA Rome Laboratories Griffiss AFB, New York			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This research developed and implemented an automated technique for translating informal specifications into formal, executable specifications. A Unified Abstract Model (UAM) was developed to combine the information contained in Entity Relationship, State Transition, and Data Flow Models into a concise, object-based representation. The UAM forms the basis for defining a formal language, the Object Modeling Language (OML), used to capture the information contained in the UAM. By using OML, we were able to develop an automated translation process to convert informal specifications into executable, formal specifications. The Software Refinery Development Environment enabled us to easily develop a parser that translates an OML specification into an abstract syntax tree. A Translation Executive transforms the information contained in the abstract syntax tree into an executable, REFINE specification. The specifier can quickly validate the correctness of the informal specification by testing its behavior. Additionally, the automatically generated executable specification serves as a basis for formal software design and future development. Two examples, a home heating system and a library database, were used to validate this formalization and transformation process. Our results clearly show the complementary nature of informal and formal methods, and provides a significant step towards formalizing the software development process.				
14. SUBJECT TERMS Automatic Programming, Software Engineering, Simulation Languages, Specifications, Computerized Simulation, Formal Methods, Object-Based Specification Language, Executable Specifications			15. NUMBER OF PAGES 350	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

Formalization and Transformation of Informal Analysis Models  
Into Executable REFINE<sup>TM</sup> Specifications

THESIS

Presented to the Faculty of the School of Engineering  
of the Air Force Institute of Technology  
Air University  
In Partial Fulfillment of the  
Requirements for the Degree of  
Master of Science in Computer Engineering

<b>Accession For</b>	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Mary M. Boom, B.S.E.E.

Captain, USAF

Bradley D. Mallare, B.S.E.E.

Captain, USAF

December, 1992

DTIC QUALITY INSPECTED 1

Approved for public release; distribution unlimited

### *Acknowledgments*

I wish to thank Major Paul Bailor for sharing his vision, giving direction and guidance when needed, and providing challenging research. Thanks to our committee members, Major David Luginbuhl and Captain John Robinson, for all their perceptive comments and suggestions, and to Dr Thomas Hartrum and Major Mark Roth for their insight. I'd also like to acknowledge the members of the Formal Methods Group and other denizens of the Signal Processing/ Graphics/ Software Engineering Lab for their support and tolerance of our somewhat eclectic working environment. In particular, thanks to Mary Anne Randour for her encouragement, her help with Lisp and REFINE, and her dynamite oatmeal-chocolate-chip cookie recipe. I also owe many thanks to my family for their support (and pretending to understand my research past the first two sentences). Lastly, thanks, Brad, for constantly pushing forward (it almost kept us on schedule) and for patiently working through all the revisions, etc.

Mary M. Boom

I owe many thanks to numerous people for supporting me during this research effort. Foremost, I would like to give thanks to the Lord Jesus Christ who gives me all things. He is *always* faithful, His strength is perfect, and it is through Him that all things are possible. My wife, Lynn, has been a pillar of strength for me. Her unselfish giving and love has inspired me more than she knows. I am truly blessed to have an *incredible* wife. I am grateful to our thesis advisor, Major Paul Bailor, who gave just the right amount of guidance to make this effort both educational and fruitful. He is an asset to the Air Force. Certainly, my thesis partner, Mary, deserves great credit. I appreciate her flexibility, hard work, and great attitude. It was fun (really!). Finally, I would like to thank our other committee members and the formal methods research group. Everyone would love formal methods if they could work with you!

Bradley D. Mallare



## *Table of Contents*

	Page
List of Figures . . . . .	viii
List of Tables . . . . .	x
Abstract . . . . .	xi
I. Introduction . . . . .	1
1.1 Background . . . . .	1
1.2 Problem . . . . .	5
1.3 Scope . . . . .	8
1.4 Assumptions . . . . .	9
1.5 Approach . . . . .	9
II. Literature Review . . . . .	13
2.1 Introduction . . . . .	13
2.2 Review of Specification Languages . . . . .	14
2.2.1 Object-Based Languages. . . . .	16
2.2.2 Algebraic and Miscellaneous Languages. . . . .	19
2.2.3 Summary of Language Assets . . . . .	21
2.3 Transforming Informal Languages to Formal Languages . . . . .	24
2.3.1 Structured Analysis to Vienna Development Method. . . . .	24
2.3.2 SADT to RML. . . . .	26
2.3.3 SADT to REFINE. . . . .	26
2.4 Conclusion. . . . .	28

	Page
III. Requirements Analysis . . . . .	30
3.1 Introduction . . . . .	30
3.2 Evaluation of Informal Models . . . . .	33
3.2.1 Data Flow Model Analysis . . . . .	33
3.2.2 Entity Relationship Model Analysis . . . . .	35
3.2.3 State Transition Model Analysis . . . . .	36
3.3 UAM Architecture Development Rationale . . . . .	37
3.4 Unified Abstract Model . . . . .	38
3.4.1 Objects . . . . .	42
3.4.2 Associations. . . . .	43
3.5 Summary . . . . .	48
IV. The Object Modeling Language . . . . .	49
4.1 Background . . . . .	49
4.2 OML Goals . . . . .	53
4.3 OML Syntax and Semantics . . . . .	60
4.4 Composing an OML Specification from an Informal Model . . . . .	67
4.4.1 How to build a specification. . . . .	67
4.5 Example Problems . . . . .	71
4.5.1 The Home Heater System Problem. . . . .	71
4.5.2 The Library Problem. . . . .	78
4.6 Summary . . . . .	84
V. Executable OML Specifications . . . . .	85
5.1 Introduction . . . . .	85
5.2 OML Compiler Generation . . . . .	88
5.3 Executable Specification Methodology . . . . .	91
5.3.1 State-Based Model Execution. . . . .	91

	Page
5.3.2 Process-Based Model Execution. . . . .	94
5.4 Translation . . . . .	97
5.4.1 Entities. . . . .	99
5.4.2 Relationships. . . . .	102
5.4.3 States. . . . .	103
5.4.4 Events. . . . .	104
5.4.5 Behaviors. . . . .	105
5.4.6 Processes. . . . .	108
5.4.7 Flows. . . . .	112
5.4.8 Stores. . . . .	115
5.4.9 Relation Table. . . . .	119
5.4.10 Constraints. . . . .	120
5.5 The Value of Executing a Specification . . . . .	122
5.6 Summary . . . . .	124
VI. Conclusions and Recommendations . . . . .	126
6.1 Objectives and Results . . . . .	126
6.2 Recommendations for Future Research . . . . .	128
6.2.1 Improvements to the Existing Translation Tool. . . . .	128
6.2.2 Additions to the Translation Tool. . . . .	129
6.2.3 Supporting Research. . . . .	130
6.3 Concluding Remarks. . . . .	133
Appendix A. Summary of OML Syntax and Semantics . . . . .	134
A.1 Syntax . . . . .	134
A.2 Semantics . . . . .	140
A.3 OML Domain Model . . . . .	142
A.4 OML Grammar . . . . .	158

	Page
Appendix B.    Ada (Subset) Program Design Language (PDL) . . . . .	166
B.1 OML with Ada PDL Domain Model . . . . .	171
B.1.1 OML Domain Model . . . . .	171
B.1.2 Ada PDL Domain Model . . . . .	179
B.2 OML with Ada PDL Grammar . . . . .	186
B.2.1 OML Grammar . . . . .	186
B.2.2 Ada PDL Grammar . . . . .	194
Appendix C.    Object Modeling Language REFINE Implementation . . . . .	199
C.1 Translation Software . . . . .	199
C.2 Utilities . . . . .	223
Appendix D.    Home Heater Problem . . . . .	244
D.1 Heater Problem Analysis . . . . .	244
D.2 Problem Statement . . . . .	244
D.3 Entity Relationship Model . . . . .	245
D.4 State Transition Model . . . . .	246
D.5 Heater Problem OML Specification . . . . .	250
D.6 Heater Problem REFINE Executable Specification . . . . .	262
Appendix E.    Library Problem Analysis . . . . .	275
E.1 Problem Statement . . . . .	275
E.2 Entity-Relationship Models . . . . .	276
E.3 Data Flow Models . . . . .	276
E.4 Library Problem OML Specification . . . . .	282
E.5 Library Problem REFINE Executable Specification . . . . .	296

	Page
Appendix F. OML User's Manual . . . . .	317
F.1 Synopsis . . . . .	317
F.2 Required Software . . . . .	317
F.3 Assumptions . . . . .	318
F.4 Generating an Executable Specification . . . . .	318
F.5 Using the Executable OML Specification . . . . .	318
F.6 Diagnosing Errors . . . . .	320
F.6.1 Errors detected while parsing. . . . .	320
F.6.2 Errors detected during compilation. . . . .	320
F.6.3 Errors revealed during execution. . . . .	322
F.7 Test Specification . . . . .	324
Bibliography . . . . .	335
Vita . . . . .	337
Vita . . . . .	338

## *List of Figures*

Figure	Page
1. Distribution of Requirements Errors by Type (9:26) . . . . .	3
2. Informal to Formal Translation Effort . . . . .	7
3. Comparison of Specification Languages . . . . .	21
4. Informal Model to REFINE translation process . . . . .	32
5. Unified Abstract Model . . . . .	41
6. Unified Abstract Model (With Referential Attributes Removed) . . . . .	46
7. OML: Bridging the Gap . . . . .	50
8. Translation Process: Informal Requirements to Executable Specifications . . . . .	55
9. Home Heater Entity Relationship Model . . . . .	73
10. Home Heater State Transition Model . . . . .	76
11. Library Problem Level 0 Data Flow Diagram . . . . .	80
12. Library Problem Level 1 Data Flow Diagram . . . . .	81
13. Steps Required for Translation versus Simulation . . . . .	86
14. State Based Model Execution Methodology . . . . .	93
15. Process Oriented Model Execution Methodology . . . . .	96
16. Hierarchy Detail with Object Mappings . . . . .	143
17. Hierarchy Detail with Object Mappings, Continued . . . . .	144
18. Hierarchy Detail with Object Mappings, Continued . . . . .	145
19. Hierarchy Detail with Object Mappings, Continued . . . . .	146
20. Hierarchy Detail with Object Mappings, Continued . . . . .	147
21. Hierarchy Detail with Object Mappings, Continued . . . . .	148
22. Hierarchy Detail with Object Mappings, Continued . . . . .	149
23. Home Heating System: Entity Relationship Model . . . . .	245
24. Home Heating System: State Transition Model . . . . .	246
25. Library: Entity Relationship Model (6:F-12) . . . . .	276
26. Library: Context Diagram (6:F-8) . . . . .	277

Figure	Page
27. Library: Level 0 . . . . .	278
28. Library: Level 1 . . . . .	279
29. Library: Level 2 (6:F-10) . . . . .	280
30. Library: Level 3 (6:F-11) . . . . .	281

### *List of Tables*

Table	Page
1. Cost to Repair Software Errors at Various Stages in Life-cycle (9:23) . . . . .	2
2. SADT to REFINE Language Mappings . . . . .	27
3. Mapping Informal Model Elements To Unified Abstract Model Elements . . . . .	39
4. Mapping OML Objects into REFINE Executable . . . . .	92
5. Excerpt from the Library Problem Relation Table . . . . .	110
6. Software Required to Support OML's Translation and Execution . . . . .	317
7. How to Generate an Executable Specification . . . . .	319



### *Abstract*

This research developed and implemented an automated technique for translating informal specifications into formal, executable specifications. A Unified Abstract Model (UAM) was developed to combine the information contained in Entity Relationship, State Transition, and Data Flow Models into a concise, object-based representation. The UAM forms the basis for defining a formal language, the Object Modeling Language (OML), used to capture the information contained in the UAM, and therefore ERMs, DFMs, and STMs. By using OML, we were able to develop an automated translation process to convert informal specifications into executable, formal specifications. The Software Refinery Development Environment enabled us to easily develop a parser that translates an OML specification into an abstract syntax tree. A Translation Executive transforms the information contained in the abstract syntax tree into an executable, REFINE specification. By testing the behavior of the executable specification, the specifier can quickly validate the correctness of the informal specification. Additionally, the automatically generated executable specification serves as a basis for formal software design and future development. Two examples, a home heating system and a library database, were used to validate this formalization and transformation process. Our results clearly show the complementary nature of informal and formal methods, and provides a significant step towards formalizing the software development process.

# Formalization and Transformation of Informal Analysis Models Into Executable REFINE<sup>TM</sup> Specifications

## *I. Introduction*

### *1.1 Background*

The elicitation and specification of software requirements is critical to the successful development of a software system. It is crucial that the user's needs and problems be well understood, analyzed, and properly documented in requirement specifications. Requirement errors can lead to a system that is over budget, behind schedule, and one that does not meet the user's needs (9:27). Primarily, software requirements are informally specified via natural language documents (i.e., English text) (7:2) and graphical models (e.g., software analysis models – data flow, entity relationship, state transition, etc.).

A study on software projects conducted by Boehm, concluded that 54% of all software project errors are not discovered until after the coding and unit testing stages. Furthermore, of these errors, 45% are directly attributable to errors in the requirements and design stages (9:24). DeMarco also performed a study of software errors and reported that 56% of all errors detected during a program originate during the requirements and design phases (9:24). These two independent studies underscore the importance of the requirements specification process. Not only are more than 50% of all software errors made during the early stages of requirements specification and design, but they also are not being discovered until late in the software lifecycle. This late discovery of software problems directly contributes to the sky-rocketing costs of software systems. Table 1 taken from Davis' textbook (9:23) shows the relative cost of fixing an error during the various stages of the software lifecycle. In this table, the repair costs are relative to the cost of detecting and fixing an

Stage	Relative Cost of Repair
Requirements	0.1 - 0.2
Design	0.5
Coding	1
Unit test	2
Acceptance test	5
Maintenance	20

Table 1. Cost to Repair Software Errors at Various Stages in Life-cycle (9:23)

error in the coding phase. Clearly, a great deal of money and time could be saved by discovering requirement errors during the requirements stage.

There are several factors that contribute to requirements errors. Figure 1, also taken from Davis' text, illustrates the distribution and types of errors made during the requirements stage for the Navy's A-7E aircraft program. As seen in Figure 1, incorrect requirements are responsible for nearly 50% of all requirement errors. These errors frequently occur when the specifier does not correctly understand the user's problem, or when users do not correctly understand their own requirements and thus do not convey them correctly to the specifier. The second highest cause of requirement errors is simply the omission of necessary requirements. Here, the specifier either captures only part of a requirement or does not capture any of the requirement in the specification. This can also be attributed to users not knowing their need for a requirement. Inconsistent requirements are the next most prevalent type of errors. These occur when two or more requirements specify conflicting information. Ambiguous requirements are also responsible for requirements errors. Here, the requirements may correctly specify the user's needs, but because of difficulties in clearly stating complex requirements in natural language, and because of diversity in reader backgrounds, these informal specifications are often misinterpreted from the user's original intentions. Lastly, misplaced requirements are also responsible for requirement errors. These errors

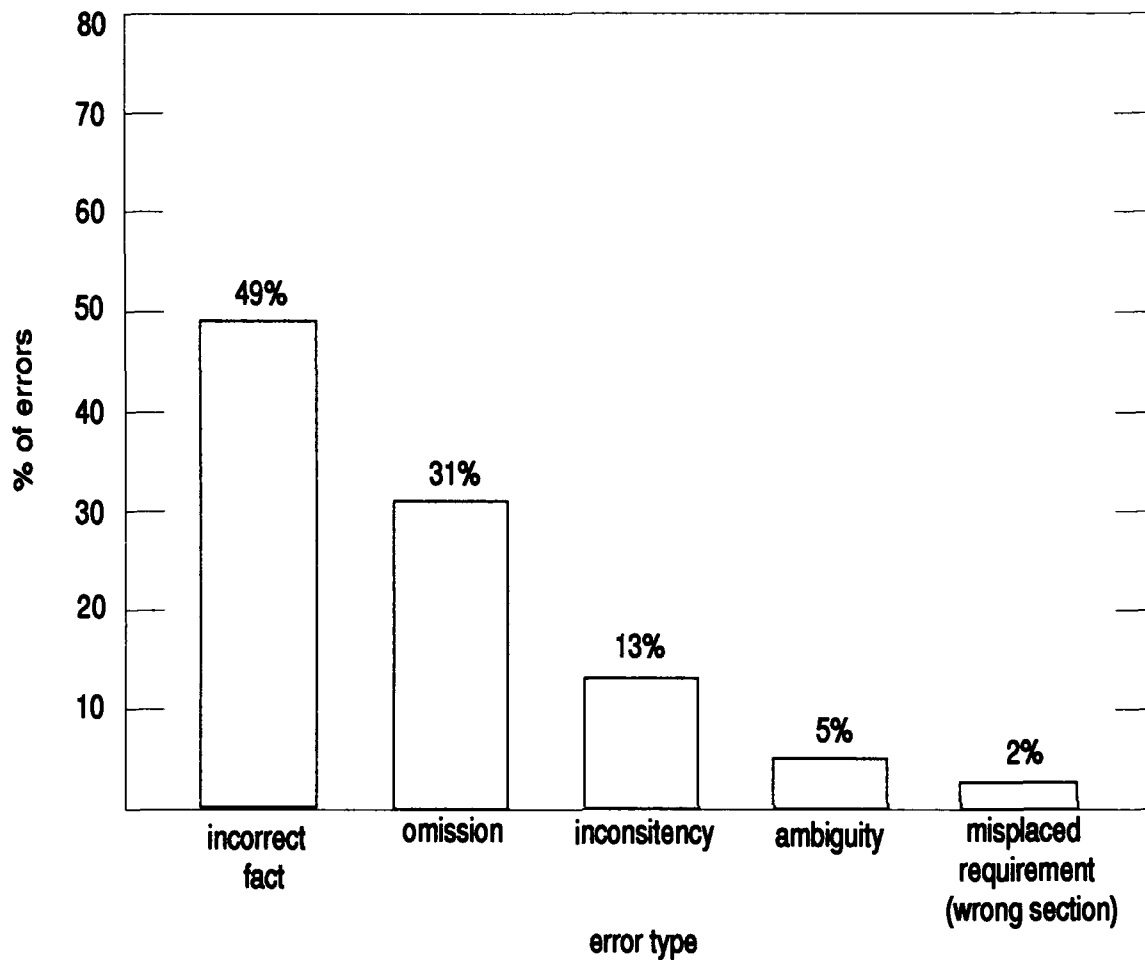


Figure 1. Distribution of Requirements Errors by Type (9:26)

are a result of necessary requirements being associated with the functionality of the wrong system components in the specification. Unfortunately, these requirement errors are often not discovered until late in development (e.g., testing) at which point (as illustrated in Table 1) they are very expensive and time consuming to correct.

In spite of these shortcomings of requirement specifications, informal specifications remain the most widely used method for specifying software system requirements. Informal specifications have several key strengths. Since informal specifications are English-like in nature and use graphical techniques, they:

- are easy to learn and understand,
- provide an ideal mechanism for eliciting requirements, and
- communicate the specifier's understanding of requirements back to the user. (13, 27)

As previously stated, however, informal specifications can be imprecise and ambiguous.

Formal specifications are another method for specifying software requirements that are gaining in popularity and respect. Formal specifications are mathematically based and possess a formal notation to model system requirements. Their mathematical nature and formal notation provide the following strengths:

- requirements are concisely and precisely specified,
- requirements are clear and unambiguous, and
- specifications are suitable for mathematical analysis. (13, 27)

Clearly, formal specifications can be used to improve the accuracy of the requirements specification process. However, formal specifications also have their weaknesses. Formal specifications are mathematically complex and require a high degree of mathematical competence. Consequently, they are difficult to learn and understand. Furthermore, because of their complexity, they are not a good mechanism for communicating with the user. Thus, formal specifications by themselves are not entirely sufficient for specifying a software system.

The overall objective of this thesis was to develop a methodology for bridging the gap between informal and formal specifications to capitalize on their respective strengths. Frequently, the two methods are viewed as competing techniques where only one method or the other can be used. In examining their respective strengths and weaknesses, however, it is clear the two methods are complementary in nature. Informal specifications are effective for eliciting requirements and communicating with users while mathematically based, executable formal specifications provide a method for resolving requirement misinterpretation, validating requirement specifications, and

serving as a basis for automated code generation. This thesis focuses on formalizing informal specification methods by using a formal language to capture informal requirements and to serve as a basis for automated translation into an executable formal specification. By converting an informal specification into an executable form, we can validate the behavior of the informal specification to uncover and correct requirement errors very early in the software lifecycle.

## *1.2 Problem*

Informal specification techniques consist of textual documents and graphical models that describe the information content and behavior of a software system. Diagrams are useful for expressing the most abstract views in informal specifications. They allow people to understand and communicate easily about large complex ideas. (29) However, even when diagrams are decomposed to show greater amounts of detail, they still must be supplemented with text to expand on abstract ideas. If written in a natural language, these textual specifications can be ambiguous.

Natural languages are extremely expressive and are often used to provide detailed descriptions needed for system specifications. However, these languages are not precise enough to ensure a unique meaning for each description. Specifications can also be misinterpreted because of the reader's or author's frame of reference. Natural language allows the specifier to make inappropriate associations between requirements and implementation-specific details. These actions should be reserved for the design stage. Also, an individual reading a natural language specification may develop an understanding of the problem in terms of his previous experience. The intent of the specification is corrupted because it has not communicated the correct information. The specification has failed its purpose of being an initial system description. Like blueprints and schematic diagrams, software specifications should provide a true representation of the planned system.

Even if a formal technique is used to concretely express software specifications, verifying the specification's correctness and completeness is still a problem. If the technique is manual, verifi-

cation can be accomplished by extensive examination and cross-referencing. This will ensure the specified system completely captures all stated requirements and is consistent with itself. Correctness can be assessed by performing partial mathematical verifications. This process is very time-consuming and does not guarantee perfect program operation unless each verified segment is totally independent (highly unlikely) or all dependencies have been accounted for in the verifications (12).

Therefore, neither informal specification nor manual formal specification techniques can guarantee that the specified system will completely or partially meet the user's expectations. However, one way of allowing the user to test whether the specification meets his expectations is to employ an automated system that allows the specifier and user to execute the specification (2).

A mathematically based specification can reduce a set of requirements to data items (objects) and their relationships. System behavior can be represented by using pre- and post-conditions, decision tables, or program design language. Use of a formal specification language, rather than English, can reduce the specifier's opportunities to introduce inconsistencies and ambiguities into the specification and to influence the specification toward a specific design or platform. An ideal formal specification captures all the detail of its informal counterpart, minimizes the chance of misinterpretation or ambiguity, and only specifies "what" must be accomplished and not "how" something is to be accomplished. A method for transforming the information contained in informal modeling techniques into formal executable specifications is needed to improve the developer's ability to build the correct product. Figure 2 depicts one way of accomplishing this transformation. This diagram is useful in visualizing the objectives of this thesis.

The objectives of our research were:

1. To establish a minimal set of constructs that represent the content and behavior of informal analysis models, specifically Entity Relationship Models (ERM), Data Flow Models (DFM), and State Transition Models (STM).

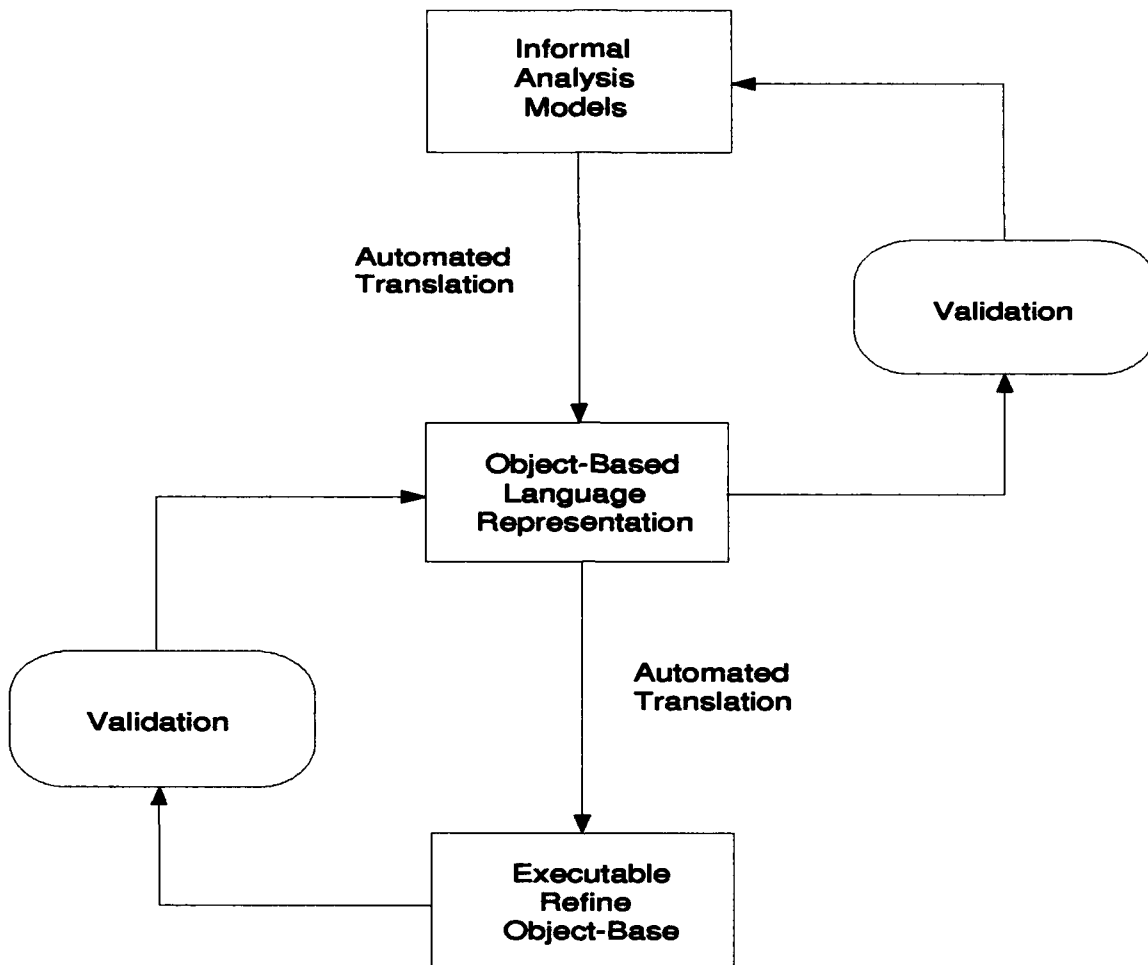


Figure 2. Informal to Formal Translation Effort

2. To develop a methodology for translating the information contained in these informal models into a formal, object-based language.
3. To develop a tool to translate formal, object-based specifications into an executable environment.
4. To validate the process of translating informal specifications into formal, executable specifications.

The REFINE Software Development Environment, developed by Reasoning Systems Inc., served as our executable environment. It is composed of a user interface, the REFINE language, and a



set of software development tools. The REFINE language, or simply REFINE, is a wide spectrum language that uses abstract constructs such as set theory, logic, transformation rules, and pattern matching. Since REFINE is an executable language and enables specifications to be expressed at any desired level of abstraction, it supports the development of executable specifications. The REFINE environment can also manipulate specifications to revise requirement specifications and initiate program development using a program transformation approach. (22:1-2)

### **Problem Statement**

To transform the information contained in informal software specifications into a mathematically based, executable formal specification that can be used to verify anticipated system behavior and can serve as a basis for formal software derivation.

### *1.3 Scope*

The primary goal of this thesis was to develop an automated process for transforming an informal requirements model, specified in an object-based language, into an executable formal specification modeled in the REFINE object-base. Since process and state behaviors can vary widely, they were represented as one or more of the following: Program Design Language (PDL), decision tables, or pre- and post-conditions. A subset of the Ada Language was used as the PDL standard for this research.

Figure 2 distinguishes two major steps for automatically translating the information represented by informal analysis models into executable REFINE specifications. Due to time constraints, we manually translated the information contained in informal analysis models into our object-based language. However, the methodology used to model the information in the object language, along with the object language's formal structure, should easily enable future automated translation. Automating this process will be accomplished in future research. An automated process was developed, however, to transform the object-based language representation of an informal specification

into a REFINE executable specification. Validation of this transformation was accomplished through test case generation and inspection.

#### *1.4 Assumptions*

This thesis was an extension of research conducted by two previous Air Force Institute of Technology (AFIT) students. Capt Randall Lee Douglass' thesis objective was to

Determine the feasibility of developing an automated mechanism that transforms a given SADT requirement analysis specification into an equivalent REFINE specification (10:4).

His results demonstrated that a manual transformation is possible between SADT specifications and their corresponding representation in REFINE. After analyzing his code, reviewing his report, and executing his test cases, it was logical to assume his transformations were correct and thus could be used as a basis for assisting in our automatic transformation. Capt Don Blankenship researched the manual transformation of several other analysis models, such as Data Flow Models (DFM), Entity Relationship Models (ERM), and State Transition Models (STM), into REFINE (6). His efforts also demonstrated that informal models can be accurately represented in REFINE. This, too, provided strong indications that an automated transformation from informal specification models into REFINE was feasible. While Douglass' and Blankenship's research both proved the value of developing executable REFINE specifications, neither effort focused on automating the process of translating informal specifications into formal specifications.

#### *1.5 Approach*

The following approach was used to reach the objectives of this thesis.

1. Conduct a literature search of currently available specification languages.
2. Analyze the information modeled by ERMs, STMs, and DFMs.

3. Develop a Unified Abstract Model (UAM) capable of modeling information contained in the aforementioned informal analysis models
4. Select a requirements specification language to support the UAM
5. Develop a process for converting the requirements language into a REFINE Abstract Syntax Tree (AST)
6. Manipulate the AST to simulate behavior of the specification and/or develop equivalent REFINE source code.

First, a detailed literature search was conducted to identify object-based languages capable of capturing the information modeled by ERMs, DFMs, and STMs. We believe that object-based languages are better suited for accurately modeling real-world problems, they promote a better description of *what* is required as opposed to *how* to meet a requirement, and they are more maintainable than functional or algebraic languages (23:ix). Representing informal models in a formal object-based specification language provided a basis for the development of an automated transformation method between the informal model and its equivalent, executable REFINE specification. Requirements analysis is still performed using informal techniques, but as a final step, all documents and diagrams are mapped into the syntax of the object-based requirements language. The object-based language sets the stage for automatic translation into the REFINE Language for behavioral analysis, and assists in clarifying any misconceptions generated by natural language or misinterpretation of diagrams. Chapter II presents the results of our literature review.

A study was then conducted to analyze the information modeled by entity relationship, state transition, and data flow models. The purpose of this analysis was to clearly identify information contained in each model, and to determine how the information from all three models could be represented in one unified model. This analysis resulted in the development of the Unified Abstract Model (UAM). Chapter III details the analysis of the informal models and describes the components comprising the UAM. The UAM, therefore, defined the components which the object-based

specification language had to be capable of representing. Once the UAM was defined, this enabled us to more closely evaluate the specification languages discovered during the literature search.

Chapter IV presents a more extensive evaluation of the specification languages. The initial literature search revealed two specification languages that appeared suitable for serving as our object-based language. However, as discussed in Chapter IV, neither of these languages was completely suitable for supporting the UAM. As a result, we developed the Object Modeling Language (OML) to fully support the UAM. OML has a formal syntax and is capable of modeling all information contained in ERMs, STMs, and DFMs. The goals in developing OML, as well as its syntax and semantics are fully described in Chapter IV. Because of OML's formal syntax, informal models represented in OML are now in a format which is amenable to automatic translation. That is, a compiler can be written to transform an OML specification into another language. In our case, we developed a compiler to translate an OML specification into a REFINE executable specification. By taking the information contained in an informal model and modeling it in OML, an executable specification can be derived to simulate the behavior of the informal model.

Chapter V focuses on the transformation of an OML specification into a REFINE executable specification. This was accomplished by a two-step process. First, the OML specification was translated into a REFINE Abstract Syntax Tree (AST). This task was supported by developing a compiler using DIALECT, REFINE's compiler generation tool. The second step required us to develop translation software to convert the information contained in the AST into a REFINE executable specification. Two example problems were then used to test the ability of OML and the translation software to convert an informal specification into a formal, executable specification.

In addition to the main text of this thesis, we have provided several appendices. The syntax and semantics of OML are presented in Appendix A, as well as the domain model and grammar to support its translation into a REFINE AST. Appendix B provides the same type of information for our Ada PDL. Appendix C contains all of the translation software required to convert information

contained in a `REFINE` AST into an executable specification. The next two appendices, Appendix D and Appendix E, present two example problems that we implemented to test OML's ability to model an informal specification, to validate the translation software, and to test the correctness of the informal specification through its execution. Finally, Appendix F is a user's manual for using the translation software to convert an OML specification into a `REFINE` specification.

## *II. Literature Review*

### *2.1 Introduction*

The primary objectives of this literature search were:

1. to determine what characteristics a "good" object-based language should possess,
2. to select a specification language as a basis for an object language, and
3. to investigate current developments in transforming informal to formal specification techniques.

There are many specification languages currently available to industry and academia, many of which are tailored to specific domains. We have classified these languages into three broad categories: object-based, algebraic, and miscellaneous. Most of the object-based languages have some basis in first-order predicate logic and were most promising to this research. They model ideas at high levels of abstraction allowing lower level details to be ignored. Many also include mechanisms for supporting classification, inheritance, specialization, generalization, etc. that support object-oriented development.

The algebraic group also has a basis in logic; however, they tend to be more functionally based than the object-based languages. Concepts are built up in axiomatic detail from very fundamental ideas, much as is done when developing an abstract data type. The high level of effort required to generate an abstract idea in this manner detracts from the clarity of such specifications. The larger the problem description, the greater this problem becomes.

The final group has been generalized as miscellaneous languages. These languages are interesting from the viewpoint that organizations have customized languages from the first two groups to develop specific solutions to real problems. This is this group's greatest weakness. Some languages reflect corporate views and ways of doing business. Others depict a specific development technique that may not be broad enough to accomplish our objectives. Others are very much like

third generation programming languages, encouraging the analyst to depart from describing *what* and to include details of *how* the problem should be solved.

## 2.2 Review of Specification Languages

A primary consideration in investigating specification languages was their ability to describe system characteristics, and to provide a logical basis for manipulation of the specification. T. H. Tse (27) and Fraser *et. al.* (13), note several desirable characteristics of requirements specification languages:

1. As a bridge between the user and the development environment, the language must be easy to employ and easy to understand by the naive user.
2. Because it serves to clarify natural language specifications, it should have a limited, well-defined syntax and semantics to describe data and technical requirements in a precise and unambiguous manner.
3. It must be suitable for both the task and the parties involved in communication. (13:455)
4. To clarify the conceptual representation of complex ideas, the language must provide a means to separate the logical and physical characteristics the specification describes, and provide a hierarchical framework to organize those characterizations.
5. The resulting specification must be modifiable and maintainable to accommodate the iterative process of requirements specification.
6. It should provide a descriptive mechanism and operators for transforming the system description from one format to another to suit different development situations.
7. Ideally, a language must support completeness, correctness, and consistency checks and proofs. (27:145)

Pamela Zave's article also provides guidance for selecting a specification language. Important issues that she emphasizes are the language's ability to model functional and non-functional requirements; the ability to support formal reasoning necessary for validation; the existence of a well-defined set of steps to construct, validate, and implement specifications; and the language's cost effectiveness. (30:212-213)

Greenspan highlights several modeling ideas he believes are essential to requirements modeling languages:

- The language must allow direct and natural modeling of the world. This is best accomplished by using an object-oriented framework where ideas and entities of the world are described using objects. Manipulation of these objects represents the behavior of the real world objects.
- It should support the organization of large descriptions. The principle of abstraction, in particular aggregation, classification, and generalization, is the primary tool for accomplishing this goal.
- It should allow the expression of assertions, entities, and activities. First-order logic is one way of meeting this requirement.
- It should uniformly use basic principles to make it easy to learn and use.
- The language's features should be precisely defined. That is, it should be formal. A formal language is based on a mathematical formalism such as first-order logic. This is necessary for the language to be well understood, well structured, and consistent. (15:3-4)

A language's expressive power sets limits on one's ability to express and reason about ideas. A language's syntax determines the ease with which a person can interpret information encoded in the language and impacts the design of any tools that are built to support the language.



### *2.2.1 Object-Based Languages.*

1. *RML*. RML addresses most of the desired characteristics and capabilities of a requirements language. RML combines knowledge-based representation concepts, object-oriented features and capabilities similar to other requirements languages. RML is built on first-order predicate logic. It has a well-defined grammar which simplifies translation from an informal to a formal language. Abstraction principles for organizing objects include aggregation, classification, and generalization. RML allows for three kinds of objects to represent real world concepts and occurrences - entity, activity, and assertion. An entity object represents things in the world, an activity object is the event that causes a change in the world, and an assertion object reflects what is true in the world. An assertion object can also describe inputs, outputs, controls, pre-conditions, post-conditions, invariants, and other properties. RML encourages the abstraction of ideas and the use of domain modeling. Greenspan's dissertation contains a complete description of RML's syntax and semantics. (15:11-26)
2. *VDM*. The Vienna Development Method (VDM) is a systematic approach to large-scale software system development pioneered by Vienna Laboratory. The method was first envisioned for the development of computer languages and their processors. However, the technique has since been applied to other systems. VDM uses decomposition and correctness arguments to specify the architecture of software systems. Abstraction is used to manage complexity. Refinements are used to transform an initial formal specification into objects that can be implemented. The method uses a language called Meta-IV to document its specifications. Meta-IV is based on first-order predicate calculus with equality (19). This provides the language with consistent, complete axiomatic definition and a set of mathematical notions that are widely understood. It includes representations for, and basic operations on, sets, maps, and tuples. It also has facilities for named and unnamed functions. Class structures are easily defined and language constructs support inheritance, although inheritance is not specifically

addressed in the syntax. Meta-IV contains most of the concepts that have been incorporated into Z (pronounced "zed"), a set-theory based language used to develop functional descriptions of computer systems, and Reasoning Systems' Software Refinery (25). Meta-IV was designed to specify systems; it was never intended to be mechanized. (5) The REFINE environment, however, implements the essential ideas of this language.

3. **REFINE.** The REFINE Software Development Environment is composed of a user interface, the REFINE language, and a set of software development tools. The REFINE language, or simply REFINE, is a wide spectrum language that provides an integrated treatment of set theory, logic, transformation rules, and pattern matching. REFINE provides much freedom to express specifications at any desired level of abstraction. (22:1-2) The specification's behavior can be evaluated by executing it in the REFINE environment. The environment also contains several valuable tools. DIALECT is a language processing tool that can be used to define grammars and read files written in the new languages into REFINE's object base. The Object Browser is a menu-driven system used to examine the static structure of the object base. INTERVISTA allows the analyst to develop graphical interfaces to REFINE. With REFINE, an analyst is also able to convert specifications from procedural structures to object-oriented implementations with minor modifications. This capability also allows the user to transform abstract specifications to more program-like specifications to enable transformation into target code.

4. *Eiffel.* Eiffel is an object-oriented programming language. It supports the ideas of class and inheritance well and uses the idea of assertions to document correctness arguments such as pre-conditions, post-conditions and invariants. (20:Appendices B-E) The language's structure is well suited for specification with respect to implicit descriptions of the domain, but the language would need to be extended to include concepts dealing with sets and maps.

5. *Spec.* Spec was developed at the US Naval Postgraduate School to be used for large scale development and to represent black-box specifications. Spec uses predicate logic to define the behavior of a model independent of its internal structure. This structure is described by modules, messages, events, and alarms. This language is different from the algebraic languages in that it is built on conceptual models rather than theories and allows the user to describe interfaces with exceptions, time dependencies, and state changes independent of the target language. Modules respond when stimulated by a message. Actions are defined by pre- and post-conditions and their associated concepts, which abstractly describe symbols in the condition predicates and help decompose the specification into manageable chunks. Spec syntax allows natural language and informal descriptions of concepts in addition to formal, mathematically-based ones. Messages define all the inter-module communication in Spec specifications. The receipt of a message is an event. Events describe the system's behavior and relate a module, a message, and a time. Alarms are discrete points in time when events are triggered and describe a temporal schedule if one is required. Spec supports time-referenced distributed systems. It also supports inheritance of concepts to ensure uniform treatment across the model. (4) Spec appears to provide a high degree of descriptiveness and structure as well as the concepts that underlie object-oriented analysis.
6. *Object-Oriented Structured Design.* OOSD is under development by Interactive Development Environments. Its notation exists in both graphical and textual forms. It is based on ideas from structure charts, Booch notation for Ada packages, class hierarchy, inheritance principles, and Hoare's monitors for concurrent programming. CASE support is being built as an extension to the Software through Pictures environment. OOSD supports a variety of design strategies. The only designs excluded are those with type or name conflicts, or with unconnected structures. The notation supports language-independent architectural design. Language-specific information contained in detailed designs must be represented as annota-

tions; there are no OOSD features to represent them. (28) Only detailed examples of the graphical notation were available.

### *2.2.2 Algebraic and Miscellaneous Languages.*

1. *PAISLey*. PAISLey is an executable specification language best suited for specifying highly concurrent, real time systems with timing constraints being the primary non-functional requirements. It is based on the data flow methodology, and generates a built-in notion of control from the calling and argument structure of the functions generated to represent DFM processes. (30:214-216)
2. *Algorithm Description Language*. ADL is an object-oriented language containing several features from Smalltalk and C++ while supporting standard third generation language features such as flow control, arrays, and string manipulation. ADL syntax is much like Pascal, contains no facilities for predicate logic, and does not have any built in functions for sets, maps, or sequences. (8)
3. *Larch*. The Larch Project has developed a family of specification languages. Each specification is written using both the Larch interface language which is programming language specific, and the Larch Shared Language which is common to all languages. These languages are algebraic in nature and do not contain abstraction concepts. (16:24)
4. *Problem Statement Language*. PSL was developed at the University of Michigan as part of the ISDOS project. Its semantics and syntax are based on the entity-relation approach. PSL supports multi-level refinement very well, allowing systems to be specified hierarchically. (27:146) However, the notions of pre- and post-conditions are not supported. Actions to be performed are specified by defining a series of steps or actions. We believe that some form of pre- and post-condition behavior description is necessary for encouraging the user to describe

his behavior in terms of *what* must be accomplished as opposed to *how* it can be accomplished.

PSL's approach to describing behavior does not discourage the user from describing *how*.

5. *EDDA*. EDDA is an attempt to add mathematical formalism to SADT. Because it is based on SADT, it cannot easily represent any other design methodology. EDDA has two forms: G-EDDA, the standard graphical version of SADT, and S-EDDA, a textual language that partially represents the graphical constructs. (27:146) From the example shown in (27), interface definitions can be expressed very clearly, but process descriptions are non-existent.
6. *Systematic Activity Modelling Method*. SAMM was developed by Boeing Computer Services Company. Like SADT, SAMM is a highly graphical representation method. It provides little support for low-level textual system specification. Documents generated by SAMM consist of graphical data descriptions showing the flow of information between processes, and process descriptions itemizing input, output, and activity requirement conditions. Specifications built with it can be analyzed using tree and graph theory. (27:148-149)
7. *Higher Order Software*. HOS is an automated version of AXES marketed by Higher Order Software Inc.. It supports a functional life-cycle model designed to support the entire development process and generate a provably correct design. The language is formal and its mathematical basis is thinly disguised from the user. (27:149)
8. *Requirements Statement Language*. RSL was developed by TRW Defense and Space Systems Group. In its textual form, it expresses requirements in terms of elements, relationships, attributes, and structures. RSL represents software by tracing the processing paths through it. Although it supports hierarchical decomposition, only the most detailed consolidated view appears in the final version of the documentation. (27:150)

Several other specification languages are also discussed in (14:Chapter 5). Because they were not directly applicable to this research, they have not been included here.

<div> <div>Languages</div> <div>Language Characteristics</div> </div>	RML	VDM	REFINE	Eiffel	Spec	OOSD	ADL	PAISley	Larch	PSL	EDDA	SAMM	HOS	RSL
Well-defined syntax	●	●	●	●	●	●	●	●	●	●	●		●	
Functional			●					●	●	●	●	●	●	●
Object-Oriented	●	●	●	●	●	●	●							
Abstraction														
Classification	●	●	●	●		●	●							
Generalization	●	●	●				●							
Aggregation	●	●	●				●							
Inheritance	●	●	●	●	●	●	●							
Descriptive	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Modifiable	●	●	●	●			●		●	●	●	●	●	●
Verifiable	●	●	●	●	●								●	
Understandable	●	●	●		●	●	●		●	●	●	●		●
Tool Support	●		●	●	●	●	●	●	●	●	●	●	●	●

● Supported  
 ● Limited Support

Figure 3. Comparison of Specification Languages

*2.2.3 Summary of Language Assets* Figure 3 contains a summary of common language features. Many languages were found to be lacking at least one attribute critical to this research; others were somewhat suited to this research's objectives. These factors are discussed below.

*2.2.3.1 Object-Based Languages.* Several of the object-based languages were suitable.

- RML encourages the abstraction of ideas and the use of domain modeling. Greenspan's dissertation contains a complete description of RML's syntax and semantics and it was a leading candidate for our selection of an object language. RML's main detracting feature was the difficulty in learning its syntax.

- Meta-IV, VDM's object language, contains constructs that represent a wide range of mathematical and logical ideas. It was an interesting option; however, its lack of mechanization and explicit inheritance capabilities made Meta-IV a less than optimal choice.
- The REFINE environment implements the essential ideas of Meta-IV. It also provides a wide range of automated transformation and inspection capabilities. It was a candidate for our object language.
- Eiffel's syntax could be parsed by DIALECT into REFINE's object base for execution and manipulation; however, it lacks representations for sets and maps. These extensions would need to be added in order to use existing tools to construct an automated environment. Purely as a language, it could be made suitable.

Other object-based languages were not as useful:

- Spec appears to provide a high degree of descriptiveness and structure, and representations for the concepts that underlie object-oriented analysis. The amount of informalism included in Spec compounds the problem of translating informal representations into formal ones and limited Spec's applicability to this research.
- The textual version of OOSD doesn't seem as rich as the graphical version. The notation has many valuable ideas; however, it is centered more on design specification rather than on requirement specification. This is a detracting feature in that it encourages the analyst to focus more on details and less on abstract domain representations.

*2.2.3.2 Algebraic and Miscellaneous Languages.* None of the languages discussed in this category were useful. The summaries below focus on what we considered to be major flaws but are not comprehensive descriptions.

- ADL contains no facilities for predicate logic and does not have any built-in functions for sets, maps, or sequences. It cannot rapidly prototype the behavior of requirements without implying design decisions.
- PAISLey is based on data flow models, largely ignoring ERMs and STMs. It was insufficient because we needed to model all three types of informal models.
- The Larch family of specification languages is algebraic in nature and does not contain abstraction concepts.
- PSL is based on an entity-relationship approach and is too limited and too specific to be used.
- EDDA only supports the SADT methodology and is too limited in scope to be useful in our research.
- SAMM lacks the semantic richness of definition needed to specify systems in the manner we intended.
- The ideas that HOS uses to generate a correct design may have been usable, but its reliance on sets of mathematical axioms and its focus on functional decomposition reduced its usefulness here.
- RSL does not support the object-oriented concepts we desired to incorporate into our specifications.

*2.2.3.3 Conclusion.* The four object-based languages listed in Section 2.2.3.1 seemed suitable as our object language and survived our initial cut. Of those languages however, *REFINE* and *RML* seemed particularly useful. We still needed to consider two important factors before making our final language choice. First, we had to consider the nature of the information that we needed to model with the language. Each analysis model (STM, DFM, and ERM) had to be examined to determine what information each model is capable of describing. The informal models also had to be considered as a group to determine if there was any overlap, redundancy,



or interdependence in the three representations. The second factor we had to consider was how the information needed to be captured in the object language. The representation must encourage the user to specify his system in an object-oriented and loosely coupled manner while facilitating an automated translation technique that is simple and direct. These issues and other important considerations are addressed in Chapter III. Another objective of our literature search was to locate current research on translating informal specifications into formal specifications. The next section discusses several groups' efforts to perform this translation.

### *2.3 Transforming Informal Languages to Formal Languages*

Frequently, formal and informal specifications are viewed as competing techniques where only one method or the other can be used to develop a system. However, completely divorcing informal modeling languages from formal languages does not take advantage of the unique benefits offered by each class of languages (13:456). One goal of this research was to represent the information contained in informal modeling techniques in an object-based formal specification language. Doing so can bridge the gap between people's mental understanding and the formal semantics. Several groups have made progress towards bridging the gap.

*2.3.1 Structured Analysis to Vienna Development Method.* Fraser, Kumar, and Vaishnavi offer motivations for bridging the gap between informal and formal specifications and two methods for transforming an informal specification into a formal specification. The authors selected a payroll system to illustrate this transformation and used Yourdon's Structured Analysis (SA) technique and the Vienna Development Method (VDM) as the informal and formal languages, respectively. These languages were chosen because they represent their respective classes, they are widely accepted in academia and industry, and they have extensive methodology support infrastructures (commonly accepted standard and notations, verification mechanisms, etc.). The first transformation method presented was a cognitive approach that uses SA modeling techniques to

guide the analyst's understanding of the system and to assist in developing the VDM specification. The second method was a rule-based approach for generating VDM specifications. (13:456-458)

The transformation method requires the problem to be informally modeled using Data Flow Diagrams (DFD). The lowest level DFD functions (functional primitives) are then described using transform descriptions represented as Decision Tables. The technique for developing VDM specifications from SA decomposition consists of a three step process:

1. Representing data flows in the data dictionary, and representing inputs from and outputs to external entities in an abstract syntax,
2. Producing a specification for each transform in the DFD,
3. Using VDM combination constructs to combine the specifications according to the architecture provided by the leveled DFD. (13:457)

The second approach is a rule-based method for interactively generating VDM specs. This approach cannot be completely automated, however, since control flows and control processes in DFDs and STDs are not required to conform to structured analysis sequencing and iteration constructs and must be manually restructured. Therefore, whenever a control flow or process is encountered the analyst must restructure them to conform to structured constructs. The automated conversion is based on three VDM conversion and composition rules. The first step consists of mapping decision table descriptions into VDM specifications using VDM's *decision table conversion rule*. Following this conversion, the specifications are then composed in a bottom up fashion using VDM's *sequence composition rule* and also its *while process composition rule*. (13:458-462)

Toetenal, Katwijk, and Plat have performed a similar transformation between SA and VDM. The authors provided a table describing a mapping from DFD constructs to VDM equivalents. A potential problem noted, however, was for most constructs there may exist more than one VDM equivalent. The authors also added the capability to modify the VDM specification in two different, but related, ways depending on the intended use of the specification. The first form

of modification is named OOFs<sup>1</sup>, and is a design method based on an object-oriented paradigm. OOFs selects components from the initial VDM specification and iteratively refines these objects into an object-oriented VDM specification that can be simply implemented into a final product. A second modification method, termed SOFOs<sup>2</sup>, transforms the initial VDM specification into a "stream-oriented" VDM specification. The SOFOs approach is a three step process which leads to an executable prototype. Therefore, depending on the developer's intentions for using the VDM specification, either OOFs or SOFOs methodologies can be applied. (26:121-126) At the time of publication, neither of these methodologies were automated.

*2.3.2 SADT to RML.* Greenspan also has described a method for translating an informal language into a formal language. Greenspan developed the formal language RML, and established a manual mapping of SADT constructs to RML. The transformation is accomplished by creating a generic object in RML for each concept defined in SADT and then specifically defining those objects in RML. RML captures the SADT structures in an object-oriented format and it uses powerful assertions to specify intended behavior and structure completely. (15:53-79a)

Each RML requirement model consists of several concept models where each SADT decomposition represents a concept model in RML. Further details for capturing the contents of SADT in RML are also addressed. Greenspan also conducted a thorough analysis of SADT to learn what information and concepts SADT contains and how to represent this information in RML. (15:53-79a) Greenspan's transformation methodology was not automated either.

*2.3.3 SADT to REFINE.* A recent thesis effort by Douglass at the Air Force Institute of Technology has resulted in the development of a methodology for translating SADT information into a formal specification language, REFINE. The overall goal of Douglass' thesis was to automate the creation of an executable specification. After evaluating the SADT language, Douglass determined

---

<sup>1</sup>expansion of acronym not provided by source

<sup>2</sup>expansion of acronym not provided by source

that the existing SADT model did not contain enough detailed behavior information to enable its translation into an executable formal specification. That is, the existing SADT model does not contain any information to indicate what inputs result in what outputs or what pre-conditions result in corresponding post-conditions. To resolve this deficiency, Douglass extended the SADT model by adding decision tables for each leaf node activity to capture the relationship between the state or value of all inputs and controls, and the corresponding state or value of all the outputs.

(10:23-25) With this extension defined, he developed the following translation technique:

1. Define a subset of both SADT and REFINE languages.
2. Develop a Common Representation to which SADT and REFINE can easily map.
3. Map the language constructs.
  - (a) Convert SADT to an SADT subset with decision tables.
  - (b) Convert SADT subset to Common Representation.
  - (c) Convert Common Representation to REFINE subset.
  - (d) Convert REFINE subset to REFINE executable. (10:25-31)

Table 2 shows how the language subsets were mapped to each other.

Language Subsets		
SADT	Common Representation	REFINE
Activities	Functions	Function
Inputs & Controls	Pre-Condition	Variable Declaration
Outputs	Post-Condition	Variable Declaration
Decision Table	Transform	Transform/Rule

Table 2. SADT to REFINE Language Mappings

Because the Common Representation to REFINE translation is so straightforward, no further conversion was necessary to obtain an executable REFINE program. The translation of decision

tables into REFINE was also simple. Each row of a decision table corresponds directly to one transform statement and all rows of each table must be mutually exclusive.

Douglass noted several benefits of this transformation technique. First, simulating the system behavior enabled him to very quickly correct deficiencies in his specifications and produce highly accurate specifications. Secondly, if his translation technique were automated, then only simple changes to the decision table are needed to automatically generate modified REFINE code. This is beneficial for generating "What if?" scenarios. Additionally, the REFINE source code can serve as the basis for design and implementation phases. Lastly, this enables any future changes to requirements to be handled by modifying specifications rather than source code. (10:41-43)

#### *2.4 Conclusion.*

This chapter has researched current literature to:

- determine what characteristics a "good" object-based specification language should possess,
- select a specification language as a basis for our object-based language,
- discover informal to formal specification translation techniques.

Figure 3 summarized the capabilities of several specification languages in terms of desirable object-based language characteristics. Several useful object languages for this thesis were identified. Chapter III provides further requirements analysis of the detailed description of data necessary for representing informal modeling techniques. This analysis clarifies which specification language format is best suited for our research. Based on our literature search, the most attractive candidates were:

- REFINE - highly descriptive and compatible with target environment, and
- RML - very descriptive and fewer degrees of freedom than REFINE.

Additionally, two informal to formal translation techniques of particular interest to us were introduced. The first technique translated Structured Analysis (Data Flow Diagrams) specifications into VDM specifications. This translation was beneficial for several reasons. First, the translation presented an example of a way to map informal information (both static and dynamic) into a formal specification. Second, DFDs, one of the three main models we will translate, were used in this translation process. Third, VDM is similar in many aspects to our target language, REFINE.

The four translation techniques described in Section 2.3 provided a significant amount of insight for us while we were developing our translation methodology. Unfortunately, none of these translation techniques (to our knowledge) have been successfully automated, and none have developed a unified model for representing the information contained in ERM, DFM, and STM. The next three chapters describe in detail the approach and *implementation* of our automated translation.

Chapter III defines a unified representation necessary for translating several informal modeling techniques (ERM, DFM, STM) into a formal specification.

### *III. Requirements Analysis*

#### *3.1 Introduction*

Software problems can be analyzed in many different ways. Three commonly used analysis tools are Entity Relationship Models (ERM), Data Flow Models (DFM), and State Transition Models (STM). These tools help define the system requirements, and form a basis for developing test cases to evaluate the system. This chapter analyzes these systems analysis models to determine the information represented by each modeling technique. The purpose of this analysis was to clearly identify information contained in each model, and determine any areas in which overlapping information is represented.

Each model can be considered as a group of data items or essential objects that describe a problem. By combining the essential objects of each model into a single model, a unified object-based model can be derived which is capable of representing the information and behavior captured by each informal model. We must also develop a methodology to express the objects defined by the unified object model in a way that can be computer-manipulated. In this case, we need a syntax defining a textual language, and a set of rules for converting an informal model into a textual (compilable) specification to express the content of an informal model in terms of the unified object model. The resulting object modeling language serves as a basis for describing an intermediate representation which simplifies automated translation from an informal analysis model into the REFINE object base. Once in the REFINE object base, the model can be exercised to demonstrate how the informally described system would behave. Execution of the specification provides a convenient means to compare the newly-generated, formal specification against the intended behavior described in the informal analysis models.

The goal of our research was to transform the information contained in informal analysis models into an executable formal specification that can be used to verify anticipated system behavior

and serve as a basis for formal software derivation. The following strategy was established to achieve this goal:

1. Analyze various informal requirement specification models (ERMs, STMs, DFMs)
2. Develop a Unified Abstract Model (UAM) capable of modeling information contained in the three informal models
3. Define an Object Modeling Language (OML) that represents the UAM in a formal, textual format.
4. Translate the OML into a REFINE Abstract Syntax Tree (AST)
5. Manipulate the AST to simulate behavior of the specification and/or develop REFINE source code.

This approach is graphically represented in Figure 4. This chapter addresses the first two steps of this process. The first part of this chapter details the data items necessary for modeling DFMs, ERM, and STMs. These models were chosen because of their wide use in the software community to specify software requirements. Analyzing these models was necessary to accurately and completely understand all information, both static and dynamic, that each model is capable of describing. DFMs primarily illustrate the functions that a system must perform; ERM represent the objects and stored data in a system, and the relationship between these objects and data stores; and STMs model the event-dependent behavior of a system (29:68-70). Each model is described in terms of the standard data items it contains. Later, these elements are represented in an object-oriented architecture required by the Unified Abstract Model and represented in its textual form, Object Modeling Language. That is, each model is described in terms of objects (essential elements), associated attributes (descriptive factors about the essential elements), and relationships between objects. This approach was selected since software designed in this manner is usually very loosely coupled and can be easily modified or expanded to include other model types.



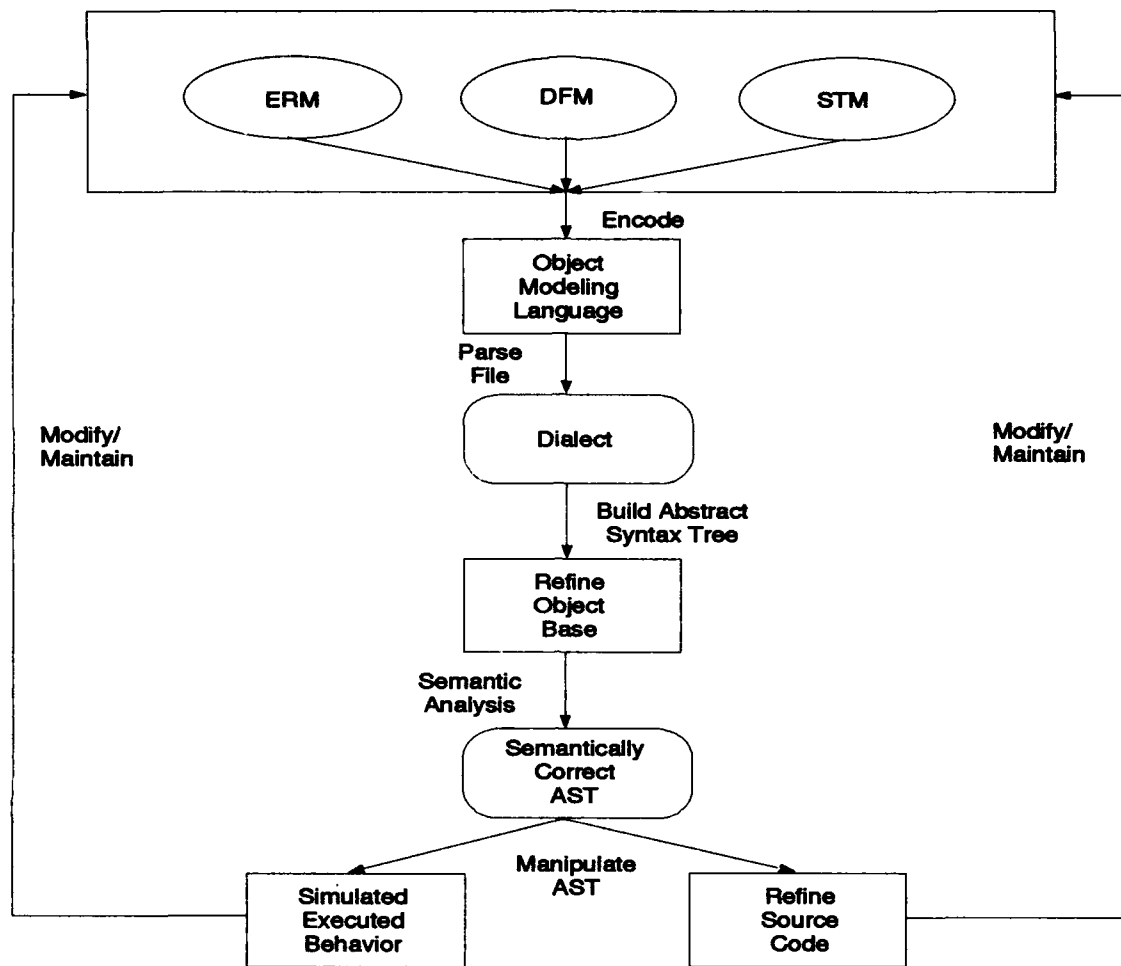


Figure 4. Informal Model to REFINE translation process

The second part of this chapter describes a Unified Abstract Model (UAM) that is capable of representing all three informal models. The intent of the UAM is to be highly general, yet robust enough to fully represent any informal analysis model or combination of models without duplication of elements. Thus, the UAM must be capable of modeling all specified data and preserving all intended behavior without adding any additional behavior. The UAM serves as the framework for defining the elements required in the Object Modeling Language (OML) (an intermediate representation) which is described in Chapter IV.

### 3.2 *Evaluation of Informal Models*

This section focuses on three popular informal analysis models: ERM, DFM, and STM. Each of these models specializes in describing a specific aspect of a system while neglecting complete descriptions of the other aspects. ERM model all the data contained in a system and provide information on how the data are interrelated. DFM emphasize the functional processes a system must perform and describe the transformation of input data into output data. STM describe the event-driven relationships in a system. (29:68-71) This section describes in detail the elements comprising each model and how the models overlap. The next section describes how they are represented in the Unified Abstract Model. The purpose of this analysis was to construct a set of elements that fully describe all aspects of a system. From this set, a minimal subset can be selected to create a unified abstract model.

**3.2.1 Data Flow Model Analysis** Data Flow Models are typically constructed using a top-down decomposition approach. Beginning with the initial system description, the specification is functionally decomposed into processes, data stores, and terminators which are linked together with data flows. Each process is abstracted to the desired level of detail until the specification provides a suitable level of understanding for both the user and the developer. A DFM may also contain control process and control flow information to model behavior present in real-time systems.

- *Processes.* Processes represent the transformation of data or the actions performed on data in a system. A process can be described by the data it uses (an input flow), the data it produces (an output flow), and the activity of the process. The input and output flows may be as large or as complex as necessary to satisfy the process behavior; however, they should be appropriate for the level of abstraction of the model. The number of behaviors needed to describe the activity of a process also depends on the desired level of abstraction. DFMs provide an excellent overview of the important functional components of a system but do not

provide any textual details on the transformation of the data. These are contained in the process specifications that accompany the models (29:68).

- *Flows.* Data flows are another major component of DFMs and they can be described as data in motion (29:143). They represent the information passed from
  - one process to another process, store, or terminator,
  - from a store to a process or terminator, or
  - from a terminator to a process or store.

A flow can be fully characterized by the data it carries, and by its source and destination endpoints (the process, store, or terminator that produced it and the process, store, or terminator that consumes it). It can be thought of as an association between two entities, although it is not necessarily a complete interface description and may not exist at all points in time.

- *Control Processes.* Control processes are part of an extension to DFMs to model the timing aspects of real-time systems. These processes act as coordinating devices that activate other processes in the DFM by sending and receiving control flows. Because of their supervisory nature, there is typically only one control process per level of abstraction in a DFM. (29:173)
- *Control Flows.* Control flows are similar to data flows; however, the information they contain is not a value. It is a flag or an indicator for a process to execute. Control flows are a way for a control process to activate a transformational process. Upon completion of the process, another control flow may be returned to the control process to notify it that the process it activated has completed. (29:172)
- *Stores.* Stores represent collections of data. Data are described by a collection of attributes: the simplest descriptions of information that can be represented in a system. Attributes are the link between the real world object and its software counterpart; that is, the target computer's representation of integers, real numbers, characters, and strings. These atomic

definitions can be grouped into higher level data items, which in turn can be grouped into sets or sequences to represent information stores. (29:149)

- *Terminators.* Terminators are entities of the outside world that interface with the system but cannot be changed by the system. They are important because they model the environment in which the system resides. (29:155)
- *Data Dictionary.* The data dictionary functions as a glossary of terms. It contains a listing of all the objects or data elements with which the system works. All composite items are expanded to show the data content and interrelationships. (29:188)

**3.2.2 Entity Relationship Model Analysis** Entity relationship models characterize the information in a system as a group of objects that are related to each other by associations or relationships. From a functional decomposition point of view, ERM's graphically depict the relationship between data stores shown in process diagrams (23:267).

- *Entities.* Entities are sets or collections of objects or concepts that exist in the real world and are of interest to the system. Each member of the collection can be uniquely identified by the value(s) of its attributes. Attributes or data elements contain information that describes all aspects of the object that are important to the system. Entities correspond to the *stores* component of DFM's. (23:271)
- *Relationships.* Relationships describe the static association between entities. They characterize important information about the way entities interact that cannot be derived from other information stored in the system. Information that must be maintained about the relationship between two entities, known as link attributes, are commonly represented by data elements (attributes) associated with a relationship. (23:32) Relationships can describe three kinds of associations: A super-type/subtype or generalization relationship, sometimes called

an "is a" relationship; an aggregation relationship, which we refer to as an "ico" (is composed of) relationship; and a general relationship.

- *Attributes.* An attribute is a single characteristic that describes a specific aspect of an object. An object can have multiple attributes that should capture all information important to that object. Also, all attributes should be independent of one another. (24:26)

*3.2.3 State Transition Model Analysis* State transition models document the dynamic characteristics of the system model in terms of states and events. Events document stimuli from outside the system, or from one state in the system to another. The response of an object to an event depends on the current state of the object. An activity is the response the object takes when an event is detected that causes a change in state. Rumbaugh differentiates between two types of state behavior: activities and actions (23:101). Activities have duration and are associated with the behavior of a specific state. They execute until they complete their function, or are interrupted by an event which causes a transition to another state. Actions are a type of behavior associated with events that occur when entering or exiting a state. Actions occur instantaneously or in so short a time that they appear to occur instantaneously.

- *States.* States are defined by the range of values that certain object attributes or groups of objects can possess at a particular time (i.e. its state space). Therefore, state represents a stage, or period of time, in the lifecycle of a system. A state is also an abstraction of an observable system activity that is waiting for some event to occur. A state has a duration and is associated with a time interval during which the system is performing some activity. (29:260-263)
- *Events.* Events are some condition or set of conditions that the system can detect. Events are considered to have no duration and can be thought of as signals that are transmitted from one object to another. Rumbaugh describes them as "all signals, inputs, decisions, interrupts,

transitions, and actions to or from users or external devices (23:173).” Rumbaugh allows the association of an action (behavior) with an event.

- *Activities and Actions.* Activities and actions capture the behavior of a system. Actions are considered to occur instantaneously, while activities have duration. STMs typically refer to several types of behaviors: entry actions, exit actions, sequences of (control) behaviors, and “do” activities. Entry and exit actions are performed on objects as a result of a change from the current state to the next. Entry actions are most commonly implemented; however, exit actions may also be needed to handle error conditions or other required actions when an event interrupts the “do” activity of a state. Sequences of behaviors represent control sequences, that is, the generation of events that cause changes in other objects (and changes in system state). Finally, “do” activities are equivalent to processes in the DFM. These behaviors can be expressed by Program Design Language, Decision Tables, or Pre- and Post-Conditions. (23:92-101)

Now that the essential elements of each informal analysis model have been identified, the next step is to combine all of these elements into a single, unified model. The next sections describes our rationale and how we have composed these model elements into a unified abstract model.

### *3.3 UAM Architecture Development Rationale*

We began our research by focusing on object-oriented languages. It became very natural for us to consider the informal models as collections of self-contained elements. Each essential element represented a component of an STM, ERM, or DFM and was evaluated to determine what attributes (or facts) were necessary for it to be fully described. To us “fully described” meant that enough information was contained in a description so that an automaton with no inferencing ability could build a useful executable specification component out of it without using any other information.

After each element was evaluated, we began looking for general similarities among the essential elements. Some informal model elements represented connections between other informal model element types. These connector elements had no definition other than to describe how two or more elements were interrelated. These elements formed our Association superclass. All other elements were grouped into the Object superclass. After making this first division, we began a detailed comparison of all the elements that we had assigned to each superclass. In the Object superclass there were several elements that represented passive objects (entities, terminals, and data dictionary entries). Because these elements were so similar, we combined them into one class (Entities) and provided that class enough attributes to fully describe any of the elements included in it. The same line of reasoning was used to combine control processes and states into one object class (States) and control flows and events into one association class (Events). Finally, we noted that several Object classes contained a behavior or action. We felt that behavior could be specified more clearly if it was described separately. The process, state, or event with which the behavior is associated only defines when the behavior will occur as the specified system executes, therefore behaviors, states, events, and processes can be described separately.

### 3.4 *Unified Abstract Model*

The analysis of the DFMs, ERM, and STMs resulted in the identification of an essential set of elements needed to completely represent each of the informal analysis models. This section combines these elements in a manner that interrelates the various models and eliminates unnecessary duplications. The word *object* is used quite often in the following sections and can have different meanings. Therefore, when speaking of the superclass, *Object* is capitalized. Any general use of the word *object* is in lower case. Table 3 summarizes the set of elements necessary to represent each model. The entries across the top are objects that exist in the Unified Abstract Model (UAM). Entries along the left column indicate the elements typically associated with each of the three analysis models. An "X" in the table indicates how each informal analysis element is represented

	UAM Object Type							
Model	Object					Association		
	Process	State	Store	Entity	Behavior	Event	Flow	Relationship
DFM								
Process	X							
Process Specification					X			
Store			X					
Control Process					X			
Control Flow						X		
Data Flow				X			X	
Terminator				X				
Data Dictionary				X				X
ERM								
Entity			X	X				
Attribute				X				
Relationship								X
STM								
States		X						
Actions					X			
Activities					X			
Events						X		

Table 3. Mapping Informal Model Elements To Unified Abstract Model Elements

in the UAM. For example, the activities associated with a STM are represented as behavior objects in the UAM. Initially, the UAM modeled all its Object types in one group. However, the elements are better conceptualized in two Object categories: Objects and Associations. Objects represent data, conditions, or activities (things represented by terminals of some sort in the informal models). Associations represent some form of relationship between two objects (things represented by arcs or links in the informal model). Association object types describe how one or more Objects are related to each other. Relationships (from ERMs) associate entities with each other, events (from STMs) associate a state with its successor, and flows (from DFMs) associate the producing and consuming processes. These three objects, therefore were grouped into the Association superclass. The remaining objects (processes, states, stores, entities, and behaviors) were grouped into an

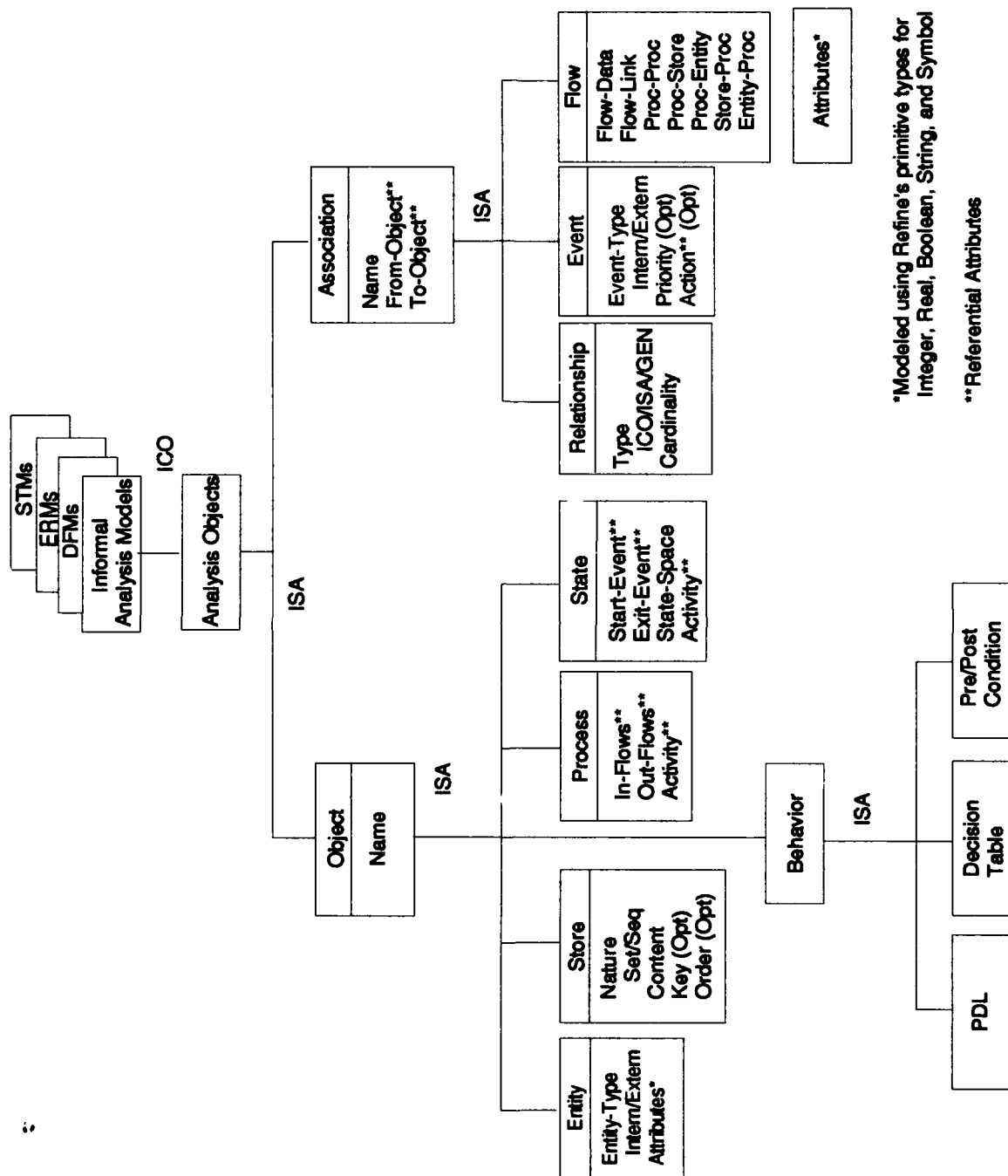


Object superclass. Association objects, therefore, represent a relationship between two or more elements from the Object superclass.

Many of the mappings between the informal model elements and the UAM objects are straightforward. However, some of the mappings require explanation. DFM control information actually represents information about the state of a system and the sequencing of events which cause state transitions (23:129). Therefore, the UAM models all process control information contained in control processes as state behavior objects, and all control flows as event objects (an association between two states). All data in a DFM is considered to be a single entity or a group of entities (a store). Therefore, a data flow is a combination of a flow association (between processes and/or stores) and an entity (the data associated with the flow). Processes are described by their input and output data flows and contain a behavior that models the process's activity. Therefore, a process in the informal model maps to a process object, a behavior object, and flow objects (representing in-flowing data and out-flowing data) in the UAM .

Primarily, the UAM serves as a template consisting of object class definitions. The only exception to this is the Entity class. Entities are used to describe many types of data that must be specifically tailored to their application. Entities are also used to describe categories of information, such as the generalized types of items that can be contained in data stores. Because they define characteristics of a group, it was more natural to allow class definitions of entities as well as instance definitions of entities.

To further define the nature of the Unified Abstract Model (UAM), the relationships between these classes needed to be defined. These relationships are graphically depicted in Figure 5. A class hierarchy was a very natural modeling method to use for this description. It shows the structure of the UAM by showing relevant objects, their attributes, and the relationships between various objects that were once parts of the three informal models. (23:21) For our purposes, decomposing the informal models into their essential elements and re-composing them into a hierarchy helped



\*Modeled using Refine's primitive types for Integer, Real, Boolean, String, and Symbol

\*\*Referential Attributes

Figure 5. Unified Abstract Model

refine our understanding of the analysis objects and develop a more abstract model that was used as a basis for the Object Modeling Language (OML). Figure 5 is an ERD showing the decomposition of the analysis objects contained in informal models. Each group is discussed below:

**3.4.1 Objects** An Object is defined as a “concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand (23:21).” It serves to promote the understanding of a problem and provide a basis for computer implementation. Therefore, all concrete or conceptual things are grouped into the Object class of the formal model. The Name attribute is a unique identifier associated with each object instantiation and is the only inherited attribute for all Object subclasses. Each subclass of Object as shown in Figure 5 is defined below:

- **State.** A state object represents the set of conditions (attribute values and relationships) characterizing the state of a system during a given period of time. A state is defined by the activity which is occurring at that time, and the set of events which cause a transition into and out of that state. Therefore, each state object is composed of attributes representing start events (which cause a transition into that state), exit events (which cause a transition out of that state), state-space (a collection of objects and the ranges of values of the objects' attributes that characterize the state), and an activity (the state's behavior). The behavior describes control activities that occur in the state, data transformation processes that occur in the state, and boundary conditions that generate events. The State-Space attribute explicitly defines which objects and attribute values constitute the state space.
- **Entity.** An entity object is defined by its attributes. Attributes are analogous to fields in a record. They allow the specifier to tailor entities to the data that needs to be specified. The Entity-Type attribute is used to specify whether the entity is internal or external to the system. To further define the entity subtypes, each subtype also possesses its own unique attributes which can be defined by the specifier. Details of how these attributes are defined are discussed in Chapter IV.

- *Store.* Store objects represent collections of data elements in a system. The Nature attribute of the store defines whether the ordering of the data is significant (a sequence) or not (a set). The Key attribute defines the entity Name attribute of an entity class that an ordered store is sorted on. The Order attribute is used in connection with the Key attribute and selects an ascending or descending order. These two attributes are used when a store is defined as being sequence-natured. The Content attribute is an entity class. The store consists of instances of this class.
- *Process.* A process object represents the functional operation of a system which transforms data. A process is characterized by its In-Flows attribute (a set of incoming flow objects), its Out-Flows (a set of outgoing flow objects), and its Activity (the behavior that transforms data in the process) attribute.
- *Behavior.* Behavior objects were added to the UAM to capture the dynamic and functional behavior of a system in a manner which is suitable for an automated translation from a specification language to an executable form. This representation must provide sufficient detail to build functioning modules that can be composed into an executable representation. The UAM provides three different mechanisms for describing the dynamic behavior of a system: Program Design Language (PDL), Decision Tables (DT), and Pre- and Post-Conditions (PPC). A subset of the Ada language defined in Appendix B is used as the PDL baseline. Any PDL with a formal notation can be used as the PDL standard. A formal notation is required, though, to enable an automated translation into an executable REFINE specification. Behaviors describe the actions performed by processes and the activities performed by states. Rumbaugh's STM distinguishes entry and exit actions associated with state transitions from the continuous behavior of the state.

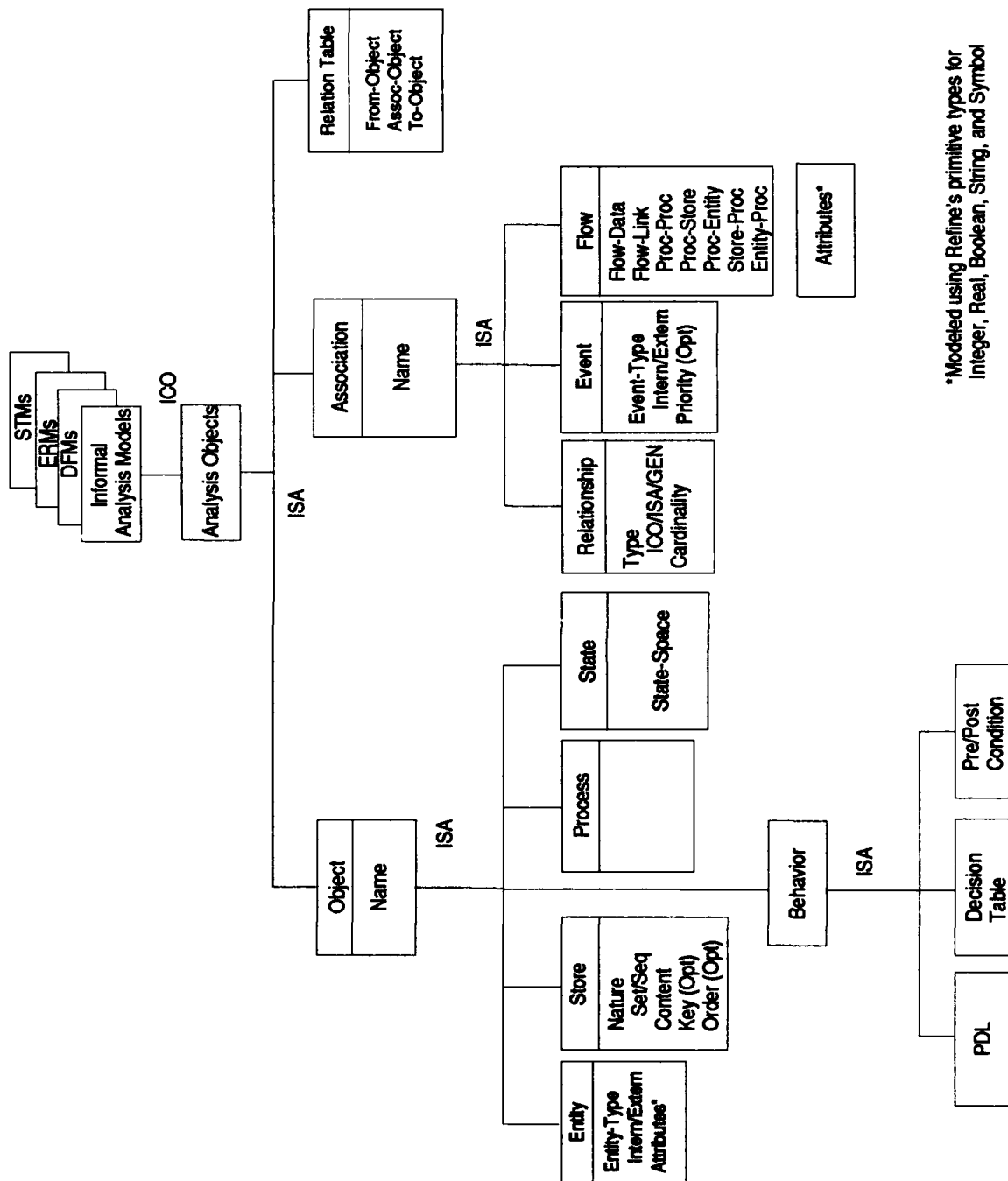
**3.4.2 Associations.** Associations represent connections between objects. They are described by a unique Name, and the objects that they associate. An association can be either bidirectional

or unidirectional. Relationship Objects show a bidirectional association between two Objects. An Event Object is a unidirectional association of two State Objects. Flow Objects are also unidirectional. The Name attribute serves as a unique identifier to distinguish Association Objects. The From-Object and To-Object attributes indicate which objects are related to each other. If a unary association is desired, then the objects are only related in the direction of the From-Object to the To-Object. If a binary association is required for a Flow Object, then two associations must be instantiated: one for each direction of association. In addition to these common attributes, the relationship, event, and flow associations have their own unique attributes:

- *Events.* An event association shows the relationship between two states of a system. The event is theoretically considered to be an instantaneous occurrence which causes a transition from one state into another. Certainly, any real action must take some amount of time; however, an event is the triggering condition which causes a state transition and is conveniently modeled as an instantaneous action. Control flows, which are usually modeled in DFMs, are modeled as events in the UAM. Control flows indicate the arrival of a specific condition or of a boolean value to a process object. Control flows do not represent data to be manipulated but rather information regarding an event which causes an activity. Therefore, a control flow can appropriately be modeled by an event object. (23:129) Event associations also require an Event-Type attribute which indicates whether the event occurs from an internal or external source. Events also have an optional Priority attribute which indicates precedence among simultaneously occurring events. The optional Action attribute is a behavior that occurs each time the event occurs. This feature has been included to keep the UAM consistent with Rumbaugh's (23) methodology. Although the capability to express actions exists in the UAM, this feature is not be used in our examples. All behavior is typically captured within the activities of a State Object.

- *Flows.* Flow associations indicate the movement of data into or out of a process object. A flow association therefore must provide a Data attribute and a Flow-Link attribute to capture this information. The Data attribute references an entity object which specifies the data elements that are in motion. The Flow-Link attribute indicates the types of the two related objects. Flow-Links are only valid between certain combinations of objects. Legal links can be formed between the following object pairs: Process-Process, Process-Store, Process-Entity, Store-Process, and Entity-Process.
- *Relationships.* Three relationship Types have been defined: Is Composed Of (ICO), Is A (ISA), and General Association (GEN). The first two types group low-level objects into larger objects and represent inherited characteristics. The third represents any other association between two objects. General Associations are typically used when describing ERM's. This appears to be the smallest set of relationship types required. Ternary relationships may be modeled by multiple general associations. Link attributes are a special case that can also be represented in terms of multiple binary relationships. The Cardinality attribute indicates the multiplicity of the related objects. Cardinality can be represented as a one to one (1-1), one to many (1-m), many to one (m-1), or a many to many (m-m) relationship.

The architecture of Figure 5 is conceptually descriptive, but many Objects contain attributes that reference other Objects. These are called referential attributes and they form an undesirable coupling between objects (23:27). This coupling makes a specification based on this architecture difficult to modify while maintaining its consistency. To eliminate this problem, the cross-references can be extracted out and recorded in a relation table with no loss in the clarity of the description. The resulting architecture is shown in Figure 6. The relation-table documents all the relationships between the other object instances contained in the UAM. It is a set of tuples that connect together two object instances and the association instance that relates them. The referential attributes were removed to a table for the following reasons:



\*Modeled using Refine's primitive types for Integer, Real, Boolean, String, and Symbol

Figure 6. Unified Abstract Model (With Referential Attributes Removed)

1. The UAM works with instantiations of the class types. In the case of many-to-many relationships, this can lead to two problems. The first is modification. If an object is modified and the many-to-many relationship changes, instances of other objects must be searched until all references to the changed object are found. If the relationship is maintained in a relation-table, the search becomes more straightforward. Second, our goal is to keep the process that we are developing platform independent. While REFINE has a means to handle many-to-many relationships, not all commercially available data base languages do.
2. The attributes of an object should be values, not other objects. This disguises the fact that the association is dependent on both objects and that it is not a part of either object by itself (23:24,27).
3. Referential attributes disguise the association as part of an object, when in fact, the association is dependent on both objects together. In many programming applications, referential attributes are represented by pairs of pointers that act as a bidirectional link. Even this is insufficient for specification because it hides the fact that the forward and reverse relationships are dependent on each other. (23:27)
4. Removing the referential attributes decouples the objects and forces the specifier to think in terms of what objects the system contains and how they are associated. As a result, objects are more self-contained, and associations are explicit.
5. Multiple cross-references make updating a specification and maintaining its consistency very difficult. The original architecture is conceptually very appealing. Representing models in this form is easy to understand, but creates a specification that has many cross-references. Specifications are judged by several criteria, one of which is maintainability (3:58). The revised architecture is slightly more difficult to understand, but yields a specification that has very low coupling and can be easily maintained. Should a specifier find this architecture difficult, it would be easy to initially model the system in the previous (highly coupled) form



and extract the revised architecture from this model. To do the reverse would not be as simple.

For these reasons, we selected the architecture of Figure 6 as the basis for the remainder of our work.

### 3.5 *Summary*

The architecture for the Unified Abstract Model has now been defined. Figure 6 provides the formal graphical notation for representing classes, objects, and associations in the UAM.

With this analysis and modeling accomplished, an intermediate language representation, or Object Modeling Language (OML), can be defined to facilitate the automated translation process to convert an informal specification into a REFINE Abstract Syntax Tree (AST). Chapter IV defines the syntax and requirements of the OML and describes the process for transforming an informal specification into an OML specification that can be compiled using REFINE. Chapter V presents our method for utilizing REFINE's AST. The first approach defines a computational process to traverse the AST and produce REFINE source code which can be compiled and executed to show system behavior. The second approach simulates the behavior of the specification by traversing the AST and executing objects in the tree based on the object's structure. Each of these methods for utilizing the information in REFINE's AST provides unique benefits to the software development process.

## IV. The Object Modeling Language

### 4.1 Background

The Object Modeling Language (OML) described in this chapter defines an intermediate representation for bridging the gap between informal software specifications and equivalent executable formal specifications. Albrecht *et. al.*, at the University of California at Berkeley, describe a method for translating one high level programming language into another. They state that an intermediate language provides a common representation to which the source and target languages can easily map. The common representation should enable the mapping to be simple, repeatable, and behavior-preserving (1:183); we developed OML to meet these requirements. OML is uniquely qualified to formalize informal specifications for the following reasons:

1. It provides a unified representation to which Data Flow, Entity Relationship, and State Transition Models can easily map.
2. It was designed to facilitate the automated translation of the above-mentioned analysis models into REFINE executable specifications.

Figure 7 depicts the importance of OML for the translation of informal specifications into formal specifications. Essentially, it serves to “bridge the gap” between informal and formal specifications. The OML, an intermediate representation, was beneficial to our research for several other reasons.

- It formalized the translation process of a source language into a target language so that the process is consistent and correct, and not affected by the skill of a specifier/analyst.
- A canonical representation facilitates an automated generation and optimization process.
- Without an intermediate representation, a unique translation mapping would have to be created for every unique problem.
- It forced a critical analysis of the modeling process. In order to design an intermediate representation language, the content and purpose of the informal modeling tools needed to

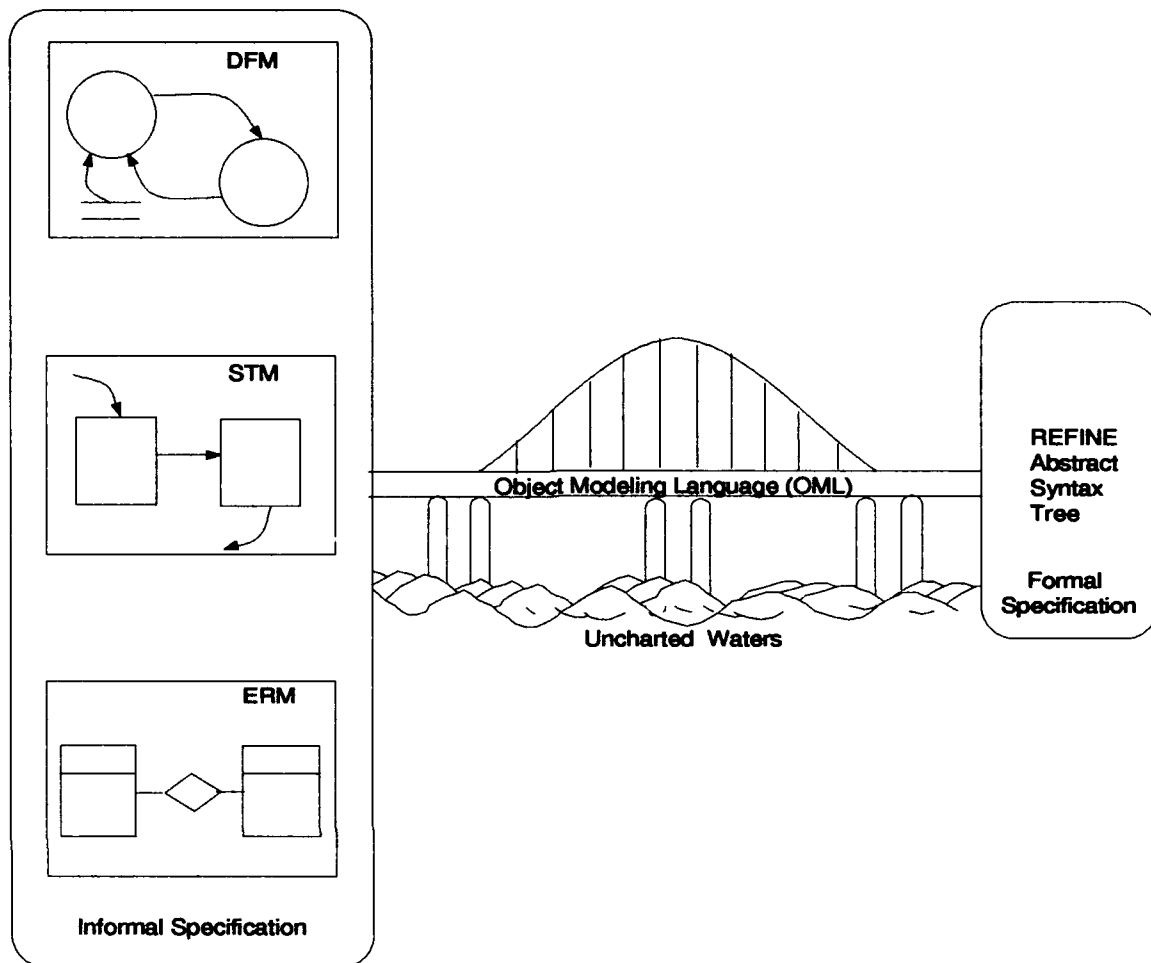


Figure 7. OML: Bridging the Gap

be analyzed. This, in turn, illuminated the general process, not just one translation effort. This is a beneficial software engineering approach.

Building a modeling language was *not* our first (nor most desirable) approach for providing an intermediate representation. Using an existing specification language was the first, most logical, approach for providing an intermediate language. In searching for a language to serve as the intermediate representation, we reviewed many currently available specification languages. Chapter II provides a summary of the specification languages surveyed. At the conclusion of our language survey, the Requirements Modeling Language (RML) (15) and the REFINE language (22) were the leading candidates for filling this role. However, after more intense scrutiny, neither of these lan-

guages were adequate for providing a unified representation of DFMs, STMs, and ERMs that can be easily translated into an executable specification.

RML was built with several beneficial modeling language principles in mind. Its mathematical basis is built on first order logic, and it incorporates several beneficial abstraction principles. RML specifications take advantage of the direct and natural modeling abilities of object-oriented decomposition, including generalization and aggregation. It also allows the expression of assertions, the description of entities, and the capturing of activity behaviors. We found all of these capabilities and qualities very important for our intermediate language. However, after studying the RML grammar and the example problems provided in Greenspan's dissertation (15), we found numerous difficulties and inconsistencies with using RML. The principles upon which RML was built and many of its constructs and capabilities have been incorporated into the Object Modeling Language (OML). However, using RML in its entirety does not satisfy our needs. The following are some of RML's problems which have been addressed by OML.

- RML does not support the direct modeling of the analysis models focused on in this thesis. In OML, we have modified several of RML's constructs to clearly and directly support a unified representation of entity relationship, data flow, and state transition models.
- RML establishes relationships within entity descriptions without fully describing the details of the relationships. It uses referential attributes to relate objects but does not capture details of the relationships such as direction or multiplicity. Referential attributes do not support object-oriented requirements for loose coupling of objects. As mentioned in Chapter III, the UAM, and thus OML, eliminates the use of referential attributes by using relation tables. Additionally, OML relationships possess a cardinality attribute to capture multiplicity, and Association class objects have implicitly defined directions (flows: unidirectional; events: unidirectional; relationships: bidirectional).

- RML is extensible. The language allows the user to specify an unrestricted set of constructs at various levels of abstraction which makes the language very difficult for automated translation. Knowledge required by an automaton to manipulate an extensible language is beyond the scope of this thesis. OML has been restricted to only allow instances of predefined OML classes, classes of Entity class objects (one of the predefined object classes), and instances of these user-defined Entity classes. These restrictions help support the translation of OML into an executable specification.
- Some RML constructs are not amenable to an automated translation, and RML does not always require the explicit definition of an object's description. RML allows the use of abbreviated constructs in its grammar. Also, when binding a value to an object's attribute, RML does not require the attribute to be specified if it can be understood from the context in which it is used. While this may not present problems in all cases, it can lead to ambiguities and performance degradation in an automated translation. The translation of context-sensitive languages into another format requires a more intense translation process. Such translations require extra and more complex searching and added semantic checking which consequently result in degraded system performance.

Defining a subset of the REFINE language was another option for specifying an intermediate representation. On the surface, REFINE was an appealing choice since it would eliminate the need to build a translator or compiler to convert an intermediate language into REFINE, and unlike RML, REFINE's grammar and syntax is robust and well-defined. REFINE is a wide-spectrum language which allows the analyst to specify problems using any combination of high- or low-level constructs. Additionally, it possesses constructs that allow functional, logic-based, and object-oriented solutions to a problem to be developed. Further, we are very familiar with REFINE's syntax and capabilities. From these perspectives, REFINE was an attractive choice. But, REFINE posed some problems

as an intermediate language which conflicted with our goal of providing a generalized translation process.

- We desired our intermediate language to be specifically tailored to the domain of the Unified Abstract Model (UAM). The domain-specific language, therefore must be capable of directly modeling the components of ERMs, DFMs, and STMs as defined by the UAM. Although the REFINE language is capable of representing informal models, it does not enforce a direct mapping into the UAM.
- We desired our intermediate language to be a generic host for translation into languages other than REFINE. By using REFINE as our intermediate language, we would be limiting the applicability of our translation process to the REFINE environment and those systems that support REFINE.

The Object Modeling Language was designed to provide the most desirable capabilities of both RML and REFINE, and to enable a unified representation of entity-relation, state-transition, and data-flow models. Many of REFINE's capabilities, such as predicate logic, set and sequence constructs, and behavioral description were incorporated into OML. The unified representation provides an intermediate form that captures the three informal models in a simple, yet explicit, manner which is easy to convert into an executable specification using an automated translation process. Section 4.3 defines the syntax, semantics, and capabilities of OML and discusses how the three informal modeling techniques are integrated into the OML architecture.

#### *4.2 OML Goals*

The investigation of several specification languages enabled us to define what capabilities we wanted in OML. However, how do we know if OML is a good specification language? Before this question can be answered, we must establish who is intended to create OML specifications, and once created, who is intended to use OML specifications.

OML specifications should be developed by a software engineer working in concert with an application domain specialist. We envision specification development to be aided by an automated elicitation tool that

- eliminates syntactic errors,
- enforces the standards of a "good" specification as defined below,
- encourages the use of object-oriented decompositions,
- avoids specifying implementation details, and
- formats requirements into a textual (compilable) file that satisfies OML's syntax and semantics.

The problem must have been analyzed in terms of ERMs, STMs, and DFMs prior to using this tool. The software engineer must be able to develop these informal models to describe the problem or be able to interpret models developed by others. He must also correctly convey the information in the dialogue with the elicitation tool as the OML specification is generated. The application domain specialist's knowledge is critical to ensure correct performance of the specification. The domain specialist is needed to ensure that the details included in the informal specification, as entered into and translated by the tool, are correct and to verify that the specification's performance will meet the user's needs. As we developed our OML specification, detailed knowledge of the intended operation of the sample problems was found in the problem description rather than the informal models. The domain specialist would be best able to provide this detailed knowledge for a real system.

Once generated, an OML specification is then manipulated by another tool that parses an OML specification into a REFINe Abstract Syntax Tree (AST) representation and then transforms the information in the AST into an executable REFINe specification. Figure 8 depicts the process of eliciting requirements to develop an executable specification. We have developed the translation

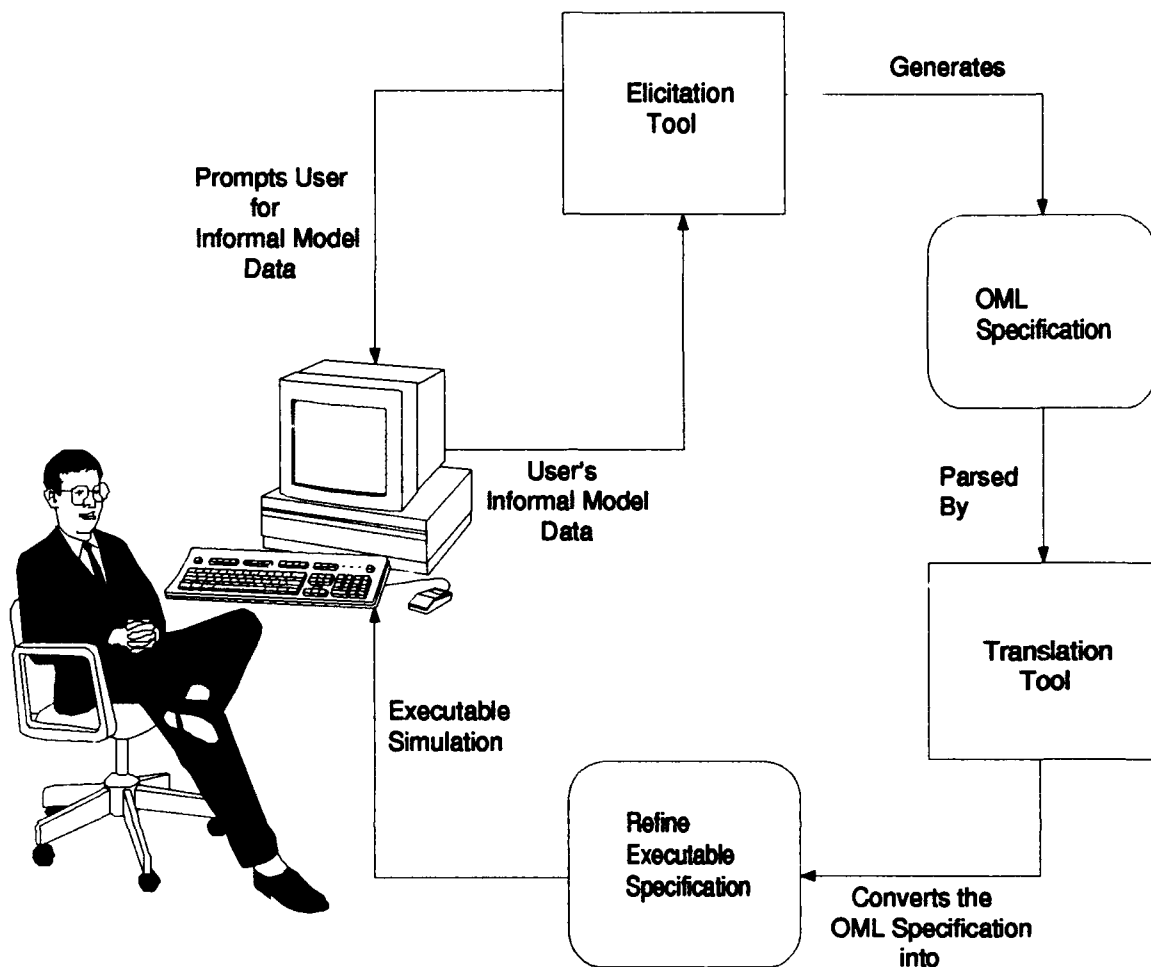


Figure 8. Translation Process: Informal Requirements to Executable Specifications

software that performs the operations required by the latter tool. The translation methodology is detailed in Chapter V and the translation software is provided in Appendix C.

Requirements can also be formatted directly (manually) into an OML specification file, as was done for this research (since an elicitation tool does not currently exist), but this is not the recommended approach. Automating the generation of OML specifications provides the following benefits:



- It will consistently generate OML specifications that are syntactically and semantically correct. This does not mean the problem has been perfectly specified. Rather, it guarantees the translation software will be able to convert it into an executable form.
- It will guide the specifier in consistently developing a good specification. Note: Our definition of a good specification is described later in this section.

OML was designed to encourage the writing of good specifications and to model specifications that possess those qualities. Since the elicitation tool must interface with OML, the builder is constrained to build a tool which enforces the way in which requirements are specified. That is, the user will be encouraged to specify his requirements in a manner that meets the standards of a good specification. However, writing a specification in OML neither guarantees that a problem is well-specified, nor guarantees that it is correct. OML's strength is that it models a problem in a manner which is easy to elicit from the user and which contains all the information necessary to facilitate its direct translation into an executable specification. By exercising the executable specification, the user can test the specification to determine if it correctly captures the desired behavior. If the executable specification does not perform as desired, modifications are simply entered into the elicitation tool and a new executable specification is generated. In this respect, OML specifications are not intended to be manually generated or directly maintained. The generated REFINES source code is also not intended to be directly maintained, but re-generated by the translator.

In addition to testing the correctness of the informal specification, the executable REFINES specification also serves as a basis for software design. As illustrated in Table 1, correcting requirement errors before the system progresses into software design (or later stages) can significantly reduce the cost and amount of time required to fix software errors. A consistent and correct specification provides a firm foundation for software design and development. A formal specification also provides the formalism necessary for the automated generation of source code. We believe that

automatic software generation will provide the same order of magnitude improvement in software production in the 1990s as compilers and high level languages did in the 1960s.

In order to evaluate the goodness of OML, it is important to understand what a specification is used for and some of the principles of good specification. Chapter II summarized several articles which addressed these issues. Balzer and Goldman (3) authored an article which addresses the criteria for judging specifications, the principles of good specifications, and the corresponding implications for specification languages. OML satisfies many of their implications for a "good" specification language. This section summarizes Balzer's and Goldman's ideas. The reader is referred to the article for a complete description of their standard.

There are three general criteria for judging specifications. These criteria are directly related to how well the specification satisfies its primary uses.

1. *Understandability.* Since the specification serves as a contract between the specifier and developer for the system to be developed, it must be clear, concise, and unambiguous. It also serves as a basis for design and implementation.
2. *Testability.* The specification is used to determine whether the developer has satisfied the contract; therefore, it must be testable.
3. *Maintainability.* The contract will change over time, thus the specification must be adaptable to change. (3:58)

The following principles of good specification ensure the criteria mentioned above are satisfied. Following the listing of these principles, OML is evaluated against each principle. One of the goals of OML is to guide and constrain the builder of the elicitation tool to enforce standards in his tool which lead to specifications that are understandable, testable, and maintainable. A good specification should:

1. *Separate functionality from implementation.* A specification is a description of *what* is desired rather than a description of *how* the desired action is accomplished.
2. *Provide a process-oriented description to capture dynamic behavior.* "Both the process to be automated and the environment in which it exists must be described formally (3:59)."
3. *Encompass the system and environment of which the software is a component.* The specification must accurately portray the system and environment with which the software interacts.
4. *Accurately model the user's view of real world objects.* The objects and operations defined in the specification must correspond to real objects and actions in the domain.
5. *Validate the results of the implementation.* The specification must be complete and formal enough to support functional testing.
6. *Tolerate incompleteness and facilitate changes.* System requirements frequently change as time progresses. The specification's structure must easily allow modifications.
7. *Localize and loosely couple objects.* Independently specified components greatly improve the maintainability of a specification. (3:58-60)

OML allows the specification of a problem that satisfies all of the above mentioned principles. The OML syntax is precise, clear, unambiguous, and behavior-preserving. The specification of each object is understandable and has only one interpretation. Behavior is described in terms of *what* happens and it captures all necessary information without adding or losing any meaning. OML's capabilities satisfy each principle in the following ways:

1. OML fosters the separation of functionality from implementation through its mathematical and behavioral description capabilities. Activity behaviors can be specified in three different ways: Decision Tables (DT), Pre- and Post-Conditions (PPC), and Program Design Language (PDL) . Both DTs and PPCs strictly support the description of *what* is desired as opposed to

*how* to obtain the desired result. Additionally, the predicate logic capabilities of OML further separate the specification of a problem from implementation details.

2. OML can capture the dynamic behavior of a problem by describing it in terms of processes or states and their associated behaviors by modeling all components of data flow and state transition models. PDL can be used to describe continuous type behaviors which DTs and PPCs cannot capture. Continuous behaviors require that operations be performed on an object over a period of time rather than at an particular point in time. For example, continuously monitoring the status of a system while other operations are occurring is a continuous type of behavior, while checking the status of a system at a specific point in time is not a continuous behavior.
3. OML's ability to model ERMs (objects and relationships), DFMs (functional transformations), and STMs (dynamic behavior) allows the description of a software problem and its associated system and environment to be consistent with informal modeling techniques.
4. OML was designed to support the object-oriented architecture defined in Chapter III. The modeling of a problem in terms of objects from the real world and relationships between those objects promotes more understandable requirements, better designs, and more maintainable systems (23:ix). Furthermore, OML has a well-defined syntax that is easy to learn and understand.
5. An automated translation process has been developed to transform an OML specification into a REFINE executable specification. The ability to execute an OML specification allows its immediate testing to determine if it specifies the intended behavior. Additionally, this formal specification serves as a basis for design, implementation, and validation testing. The transformation process is described in Chapter V.

6. Requirement modifications are easily accommodated in OML. Since an automated translation process exists to transform an OML specification into an executable specification, the effects of requirement changes can be immediately seen.
7. Two architectures for OML were described in Chapter III. The first architecture modeled objects with referential attributes to show relationships between objects. This architecture was refined to reduce its strong coupling between objects. The final architecture on which OML was built eliminated all such coupling by removing referential attributes and by describing objects in their purest form with object relationships modeled in a relation-table.

#### 4.3 OML Syntax and Semantics

This section summarizes OML's syntax requirements and several important design decisions made while defining the OML grammar. The complete definition of OML's grammar in Backus Naur Format (BNF) is provided in Appendix A. As described in Chapter III, an OML specification is created by defining a set of Analysis Objects (see Figure 5, page 41) which specify the problem. The objects that can be instantiated are entities, stores, processes, states, behaviors, relationships, events, flows, and relation-tables. The attributes associated with each of these objects are shown in Figure 6 and are also delineated in the OML grammar, Appendix A. In addition to incorporating the principles mentioned in the previous section, we made several other design decisions concerning the OML grammar which have simplified the processes needed to automatically generate and translate OML:

- *Require as little information as possible in the specification without sacrificing clarity and losing critical information.* This minimalist attitude was responsible for simplifying the OML syntax. For example, many languages such as Ada require the declaration of variables before they can be used. In OML, variables are declared and used when they are instantiated in Entity Objects. Requiring a minimal amount of punctuation was also desired; however, in

order to eliminate parsing errors, a certain amount of punctuation was included. Only three types of objects require end of line punctuation. If more than one user-defined attribute is given for an Entity object, then each attribute definition must be separated by a semicolon. Similarly, each pair of state space variable names and values must be separated by a semicolon. Lastly, all rows in a decision table and all pre-post-condition statements must be separated by a semicolon.

- *Developing a specification which is easy to parse and translate was more important than developing a specification that is visually appealing.* Since OML specifications are intended to be generated by an automated tool, the convenience of manual generation was not of great concern. We were more concerned with developing a language which allows a straightforward process for creating and translating OML specifications. Even though we were not concerned with the manual generation of OML specifications, this does *not* constitute a license for the specification language to specify constructs incompletely or inconsistently. On the contrary, all uses of punctuation, keywords, and definitions are written in a consistent manner.
- *Optimizing the size of the specification was not a concern.* OML's grammar could have been modified to simplify certain constructs and allow more compact specifications, but this would have added extra complexity to the grammar and added difficulty in translating the specification. Optimizations can also reduce the clarity of a specification which can lead to ambiguities. Optimizations of these kind are not an appropriate concern for a specification language. This is best reserved for the actual implementation. (3:58)
- *Capture system behavior using Decision Tables, Pre- and Post-Conditions, and Program Design Language.* Capturing the correct behavior of a problem in the specification was a significant hurdle when defining the grammar. There are two general types of behavior that must be captured: control activities and transformational activities. Control activities define the sequencing of activities or events which must take place in a system. These activities

are generally considered to execute instantaneously although they actually execute in a very short amount of time. Transformational activities occur over a period of time, cause a state change to occur within the system being specified, and can be thought of as procedures or functions in a high level language such as Ada. OML provides three methods for capturing these behaviors: Decision Tables (DT), Pre- and Post-Conditions (PPC), and Program Design Language (PDL). DTs and PPCs are excellent for capturing *what* the desired behavior of a system is without specifying *how* to accomplish that activity. PDL is suitable for capturing desired behaviors which are too complicated to capture in either DTs or PPCs and must be limited to these situations.

- *Variable declarations must be data typed.* As a rule of thumb, a specification should not impose any implementation details. One example is not requiring the user to specify the data type of variables or object attributes (i.e., integer, real, boolean, etc). However, to convert an OML specification into an executable REFINE specification requires that variable types be defined. Perhaps it is possible to extrapolate the data types of variables from other information in the specification, but this requires a level of knowledge base sophistication which is beyond the scope of this research.
- *Object Classes can only be created for Entity Objects.* Modeling data elements using classes and instances is a natural way for modeling data and is very important for conceptualizing a problem. Entities, in an ERM, take advantage of such structuring of data. For this reason, Entity objects can be created as classes or instances of a class. Many times during informal analysis, other types of objects such as states and relationships are organized into classes and instances of classes. For the purposes of developing an automated translation, we have not seen a need for specifying hierarchies of these types of objects since all necessary behaviors are usually specified at the leaf nodes (e.g., instantiations). If, at a later time, this design decision is not sufficient for all applications, then a class structure can easily be added to the OML grammar in a manner similar to that for Entity objects.

- *User-defined attributes can only be defined for Entity Objects.* Entity objects are the only objects that have user-defined attributes in addition to those predefined in the OML architecture. Since Entities are used to describe real-world objects, the OML must be flexible enough to characterize all required aspects of these objects. User-defined attributes are used to describe these aspects. On the other hand, other Analysis Objects do not need as much descriptive flexibility as Entity objects and cannot be augmented with user-defined attributes. If descriptive deficiencies are found, additional mandatory or user-defined attributes can be added to the grammar. The difficulty we expect to encounter if other Analysis Objects are allowed to have user-defined attributes is how those attributes will tie into the execution of the specification, and how the translation software will tie them in.
- *One or more relation-tables can be created.* All relationships can be captured in one relation-table or broken down into several different relation-tables. Creating one universal relation table is easier to build; however, multiple tables can also be useful. From an automated tool viewpoint, multiple relation tables can improve searching speed and modification efficiency if implemented properly. For example, making a relation-table for each kind of Association Object (Flow, Event, or Relationship) can simplify the search for a specific relationship. From a human understanding viewpoint, separate tables will improve readability and understanding.
- *Decision Table rules are represented by the columns.* Decision tables can be constructed in several different ways. We selected the format recommended by Hurley (17). In this format, the table can be envisioned as four quadrants. The top half of the first column in a decision table represents the first quadrant. The entries in this column identify the pre-condition variable names for each rule. The bottom half of the first column represents the second quadrant which defines the post-condition variable names associated with each behavior rule. Each remaining column in the table represents a behavior rule. The top half of each subsequent column defines the values for each pre-condition variable, while the bottom half of the column stipulates the post-condition values associated with the rule. If the pre-



condition variable values in the top portion of a column are satisfied, then the values in the bottom portion of the table are assigned to the post-condition variables. In OML the top and bottom portions of a decision table are separated by the symbol “- ->”. The illustrative problem at the end of this chapter provides an example of a decision table.

OML has the following semantic requirements:

- *All analysis object names must be unique.* This requirement was levied to simplify the process of transforming the format of variable names in behaviors. Entities and events are referenced and operated on in the behavior objects. The object names could be made unique by the translation software, but this would make resolving all the references in the behaviors more difficult. For example, if an entity and a store had identical names, the translation software would assign a unique name to each object. While translating the behaviors, the translation software would have to transform the references to objects by their *context*, or rely on human intervention whenever a name could map to more than one possible unique name.
- *An entity object's range field defines the attribute's legal range of values.* This requirement is needed for static and dynamic constraint checking, and to assure that the state-space is fully defined.
- *The first state in the OML specification is assumed to be the start state.* Rather than providing additional syntax to specify a start state, we felt that this positional notation would be a natural representation of the start state. A provision needs to be included in the requirements elicitation tool to identify an initial state.
- *The initial values assigned to all entity attributes must satisfy the state space of the initial state.* In a state-based model, initial values must be defined when specifying entities. These initial values must be in the domain of the start state (the first state specified). When the specification is executed, the initial state space is verified. Incorrect specification of initial

values will cause a premature termination of execution. To keep the specification process unambiguous, we intentionally omitted any mechanism for defining default values.

- *The data variables used in the expressions and statements in behavior objects must be attributes of entity or flow objects. The entity attributes must be fully referenced by giving both the object name and the attribute name (e.g., object-name.attribute-name). The arguments used in defining each state's state-space must refer to existing entity attributes.* A fully specified system will have its state-space defined. The state-space is described in terms of the objects in the state-space and the values of those objects that are pertinent to any particular state in the system. This requirement merely states that the specifier must declare all the objects on which any state in the system depends in order to declare the state-spaces of the individual states. Therefore, all events and entities should be defined prior to states and behaviors so that state transitions defined in the behaviors can be made based on objects that exist.
- *Each state's state-space definition must include the value or range of values of all object attributes that are important to the state.* OML has no implicit means of including values for entity attributes in a state-space. Therefore, all objects required to have certain values must be included in the state-space definition of the state using OML expressions and "dot" notation. Because the state-space is verified each time the state is entered, this requirement is useful to verify that events and behaviors that modify the entity values are performed in a manner that is consistent with each state's specified state-space.
- *The key and order attributes only apply to sequence-natured stores.* Because sets are unordered groupings that have no duplicate members, key (the field the store is ordered on) and order (ascending or descending) are meaningless (and therefore ignored) during the translation of set-natured stores. A more thorough discussion of this feature of Stores is given in Chapter V, Section 5.4.8.

- *A flow-object's flow-data attribute requires the name of the entity class of the data that will be carried by the flow-object.* This provides the translation software with an entity class (that indicates the type of data the flow will carry) to instantiate when creating the flow-object .
- *External events must be associated with a behavior object to cause a change in state-space object values.* External events represent changes in the real-world state space (caused by objects external to the modeled system) that the system must detect. The behavior associated with an external event changes the values of entity-attributes causing a change in the system's state space.
- *The ICO association used in relation tables is reserved for relationships between a process, state, or external event and its behavior.* ICO allows the translation software to match the appropriate object with its behavior to compensate for the elimination of referential attributes from the object definitions.
- *The event field in behavior objects will only be used by state behaviors and control process behaviors. External event behaviors cannot specify next events.* State and control-process behaviors contain information that tells the control architecture what state or process should be executed next. These are the only behaviors modeled in OML specifications that have controlling characteristics, because they must model all the decision-making capability of a state or control process. All other behaviors model transformational activities only and cannot contain the directional capability that is modeled by the event field.
- *If multiple behaviors are associated with state objects, they must be listed in order of execution in the relation table. Each behavior will be executed in this order and the last state behavior should specify a next event.* If the "do" activity of a state requires that a series of transformations occur prior to the execution of a state transition, this can be modeled using multiple activities. Because the control architecture cannot yet deal with multiple events at one time, we had to restrict the number of events that were returned to the controlling func-

tion. Events cause a change in the flow of control. The controlling function of the simulation routine assumes that a state transition is being made. No facility exists for the controlling function to recognize a nested state or to determine whether it should return to a previous state after several transitions.

- *All external events, processes, and states must have behavior objects.* These objects explicitly state what actions should occur during a state or process, or what state-space attributes change as the result of an external event.
- *Set operations have been incorporated into OML.* In the following discussion, S and X are sets and x is an element of a set:
  - The *set-diff* operation requires two arguments, both of which are sets (e.g., S set-diff X). This operation removes the elements of the second argument from the first argument.
  - The *union* operation requires the first argument to be a set and the second argument to be an element (e.g., S union x). This operation adds the second argument to the first argument.
  - The *in* operation requires the first argument to be an element and the second argument to be a set (e.g., x in S). This operation returns true if the first argument is in the second argument.
  - The *getitem* command locates a specific item in a store and allows the item to be modified, but does not remove the item from the store.
  - The *getset* command returns a set of items in a store but does not remove the set from the store.

#### 4.4 *Composing an OML Specification from an Informal Model*

4.4.1 *How to build a specification.* In order to represent a problem in an OML specification, the user must have already abstracted the problem using informal analysis models. Frequently,

when analyzing a problem, all three informal models (entity relationship, state transition, and data flow models) are needed to capture all of the information required to solve a problem. However, for some applications, it may not be necessary to use all three models. For example, it may be possible to fully specify a problem by only using two of the three informal models. As discussed in Section 4.5.1, we were able to completely model the Home Heater System Problem by only using ERMs and STMs. Similarly, our analysis of the Library System Problem (Section 4.5.2) only required us to develop an ERM and a DFM.

The order in which these models are created is influenced by different modeling approaches. The functional (structured analysis) approach, as described by Yourdon (29), places primary emphasis on the functionality of the system. Using this approach, DFDs are created first, followed by STMs, and then ERMs. The object-oriented modeling approach presented by Rumbaugh (23), places more emphasis on first identifying the objects in the application domain and then defining the functions that act on those objects (23:7). Using this methodology, ERMs (object models as per Rumbaugh) are created first, followed by STMs (dynamic models), and finally DFMs (functional models). We strongly advocate the object-oriented approach to modeling a system. An object-oriented description of the objects and relationships in a system is very useful for modeling real world concepts and improving understanding. The reader is referred to Rumbaugh's text (23) on object-oriented analysis and design or Yourdon's text (29) on structured analysis for guidance in developing these models.

It is also important to understand how the three models interrelate. The ERM forms the groundwork for the other two models. Therefore, the STM and DFM are related to the ERM and each other in the following ways:

- The activities and actions of an STM are represented as processes in the DFM.
- Flow objects in the DFM correspond to entity objects in the ERM. (23:179)
- State and process behaviors only use entity attributes defined in the ERM.

- The STM shows the sequence in which processes are performed. (23:138-139)
- The entities in the ERM become stores in the STM.
- Control flows in the DFM are represented as events in the STM.
- Processes accept and send entity classes defined in the ERM. (24:97)

Until the elicitation tool is developed, OML specifications must be manually generated. Therefore, the development of OML specifications is of interest. Although there are no restrictions on how OML specifications can be created, we recommend that the developer begin by specifying all data objects. In an object-oriented decomposition, these objects will be modeled by ERMs. If a functional approach is used, this implies that the data described in a DFM and its accompanying data dictionary should be represented first. It is important to represent data objects first since state descriptions and data transformations are meaningless unless there are objects to be manipulated. Once the data objects have been defined, the order in which other OML objects are created is not critical.

Since OML specifications will be generated by a requirements elicitation tool, the following discussion is intended to provide guidance in its development. The syntax and semantics of OML are specified in Appendix A.

- The elicitation tool should save information in any convenient form and should be capable of generating an OML text file. The OML specification is simply a flat file representation of the informal analysis models.
- Instantiate all Entity and Relationship objects necessary for representing the problem's ERM. The user can create an Entity class (set of entity instances) or an Entity Instance. The user can then define attributes to describe the entities. Every attribute must be given a range of valid values and an initial value. Whether these values are defined in an Entity class, in an Entity instance, or in a combination of both is up to the user. Once the Entities and

Relationships have been created, then create a Relation-Table object and add the relationships between these objects into the table. These relationships are bidirectional.

- Represent the STM in OML. First, define the State objects in the system. The state's state-space attribute must define the values or ranges of all important entity-object attributes which characterize the state. These attributes must be referred to by their full name. For example, the Status attribute of a Motor entity instance would be referenced by *motor.status*. After defining States, create the Events which cause a transition between two States. Events, by definition, are unidirectional. Next, create the Behaviors associated with each State. When state behaviors are being described, a next event must be identified with each control path of the behavior. This is necessary to direct the sequencing of states. Variable names used in behavior descriptions must exist as Entity attributes. Once these three object types have been defined, add entries into the Relation-Table to associate all the objects (i.e., relate states with events, and states with behaviors).
- Represent the DFM in OML. Objects should be created in the following order: Process objects, Flow objects, Entity objects, Store objects. Process objects are characterized by their behavior. Therefore, Behavior objects must be created and associated with a Process object in the Relation-Table. Flow objects are characterized by the data (entities) they transfer. Accordingly, associations must be entered into the Relation-Table to relate Flows with Entities. Stores are represented by a set or sequence of Entities. For this reason, Store objects must indicate whether ordering is important, and if so, define a key to sort on.
- Save and compile the file. Before a file can be compiled, the REFINE environment must be running and the translation software must be loaded into the REFINE environment. Details on starting REFINE and loading the translation software are included in Appendix F. To parse an OML specification, type the following command at the REFINE prompt:

```
(parse-file(your-file-filename,  
            false,  
            find-object('re::grammar, 'oml),  
            find-object('re::grammar, 'oml),  
            find-package("RU") ) )
```

The next section discusses how two sample problems were implemented in the Object Modeling Language.

#### *4.5 Example Problems*

Chapter III defined the mapping of informal model elements into OML objects. Previous sections of this chapter defined the syntax and semantics of OML and also the process for creating an OML specification. This section implements two sample problems in OML. They were chosen for the following reasons:

1. To validate the effectiveness of OML as a specification language.
2. To serve as an example of how a problem is specified in OML.

Testing the ability and flexibility of OML to represent different types of problems is one way of validating the Object Modeling Language. Additionally, since the OML specification will be created by an automated tool, the following problems serve as examples of the type of output the tool must be capable of generating.

The first problem, the Home Heater System, is a state-based problem and is conveniently modeled using ERMs and STMs. The second problem is the Library System which is a static, data storage problem and is represented by ERMs and DFMs. Because of their differing nature, these problems were chosen to test the ability of OML to represent both dynamic and static oriented problems. Both problems were taken from the Fourth International Workshop on Software Specifications and Design (18).

*4.5.1 The Home Heater System Problem.* The Home Heater problem can be accurately specified by an Entity Relationship Model and a State Transition Model. A Data Flow Model



can also be used, however in this case, it only represents redundant information. The complete problem statement, an analysis of this problem, and the complete OML specification are provided in Appendix D. With some modifications, the state transition and entity relationship models developed by Blankenship are used for this analysis (6:Appendix C). Regardless of whether the specifier chooses to analyze this problem in an object-oriented or functional fashion, the problem can be easily translated into OML.

The Heater problem describes the furnace system components and the sequence of control events necessary for regulating the air temperature of a home. When specifying almost any problem, the ERM should first be translated into OML. Figure 9 depicts the entity relationship model for the Heater problem.

Begin by converting the entities in the ERM into OML entity objects. The following example converts the OIL-VALVE and WATER-VALVE entities into OML entity objects. The two valves are grouped under a general class of valves. This is done by creating a VALVE object class and by making the OIL-VALVE and the WATER-VALVE instances of the VALVE object. In this case, the OIL-VALVE "is a" (instance of) VALVE and the WATER-VALVE "is a" (instance of) VALVE. This approach is not necessary for specifying this problem; however, we've modeled the valves in this manner to demonstrate how object classes and instances can be created.

As defined in the OML grammar, two types of entity objects can be instantiated: entity classes and entity instances. The user has the freedom to specify any attributes which define that object. Attributes defined in entity class objects are inherited by all subtypes of that class. If attributes are desired at the class level, they can be specified either by a full attribute definition, or by an abbreviated attribute definition. A full attribute definition requires the attribute's name and type, its range of legal values, and its initial value to be declared. An abbreviated definition can be used in the class definition by declaring the attribute's name, type, and range of legal values. In the second case, the initial value is specified when the object instance is created. Attributes can be

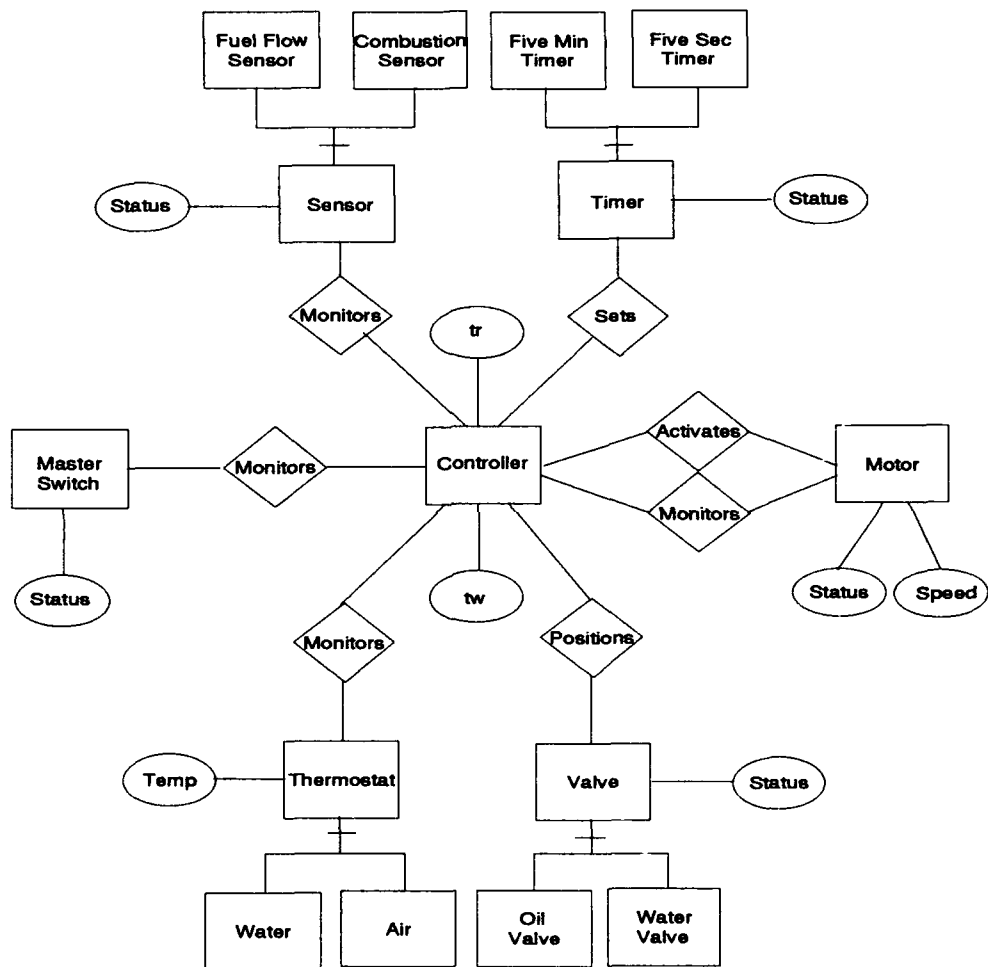


Figure 9. Home Heater Entity Relationship Model

declared as any of the following types: integer, real, boolean, string, set, or sequence. The ranges can be specified either by enumerating each possible value or by giving a start value and an end value. Ranges are not required for boolean or string type attributes.

For this example, a valve class and two valve instances must be created. The OML code that corresponds to these objects is:

```

VALVE class-of entity
  type : external
  parts
    status : symbol range {open, closed}

OIL-VALVE instance-of VALVE
  values
    status : closed

WATER-VALVE instance-of VALVE
  values
    status : closed

```

Since both valves have a *status* attribute, we chose to declare the *status* attribute in the class declaration and to define initial values at each instance declaration. As another option, since the OIL-VALVE and WATER-VALVE have the same initial status, the initial value can be defined in the class definition as shown below.

```

VALVE class-of entity
  type : external
  parts
    status : symbol range {open, closed} init-val closed

OIL-VALVE instance-of VALVE

WATER-VALVE instance-of VALVE

```

Once all the Entity objects are created, the next step is to create the Relationship objects which relate two entities together. Referring to Figure 9, both valves are related to the controller object by a POSITIONS relationship. Since the definition of a Relationship object does not force its coupling to any specific entity objects, the same POSITIONS relationship can be used to relate both the OIL-VALVE and the WATER-VALVE to the CONTROLLER. Thus, the OML code which models this relationship is:

**POSITIONS instance-of relationship**  
**type : general**  
**cardinality : 1-1**

The remaining relationships are created in a similar manner. The POSITIONS relationship object and the VALVE objects have not yet been associated together in the OML file. Associations between objects are captured in a Relation-Table. Once the relationship objects are created, the entity objects and relationship objects should be associated together in a Relation-Table. The VALVES and CONTROLLER are associated together in the following abbreviated table:

**TABLE1 instance-of relation-table**

CONTROLLER,	POSITIONS,	VALVE;
CONTROLLER,	MONITORS,	MOTOR;
CONTROLLER,	MONITORS,	THERMOSTAT

Note that all rows of a relation-table end with a semicolon *except* for the last row. Once these associations have been entered into the relation-table, all the objects necessary for representing an ERM in OML have been defined. The next step is to specify the STM.

Figure 10 illustrates an STM for modeling the Heater problem. In order to represent an STM in OML, State, Event, and Behavior objects must be created.

The State objects should be created first. The MOTOR-ON state of Figure 10 is provided here as an example. This particular state assumes that certain entities such as air, motor, ignition, and OIL-VALVE have been previously declared, since the state-space references the attributes of these objects.

**MOTOR-ON instance-of state**  
**state-space :** MASTER-SWITCH.status = on;  
AIR.temp < CONTROLLER.tr - 2;  
MOTOR.status = on;  
MOTOR.speed = inadequate;  
IGNITION.status = off;  
OIL-VALVE.status = closed

The state-space attribute must define the values of all objects that characterize the system during that state. Notice the state space is composed of previously defined entity attributes and specific attribute values. The entity attributes should be referenced by using the "dot" notation which

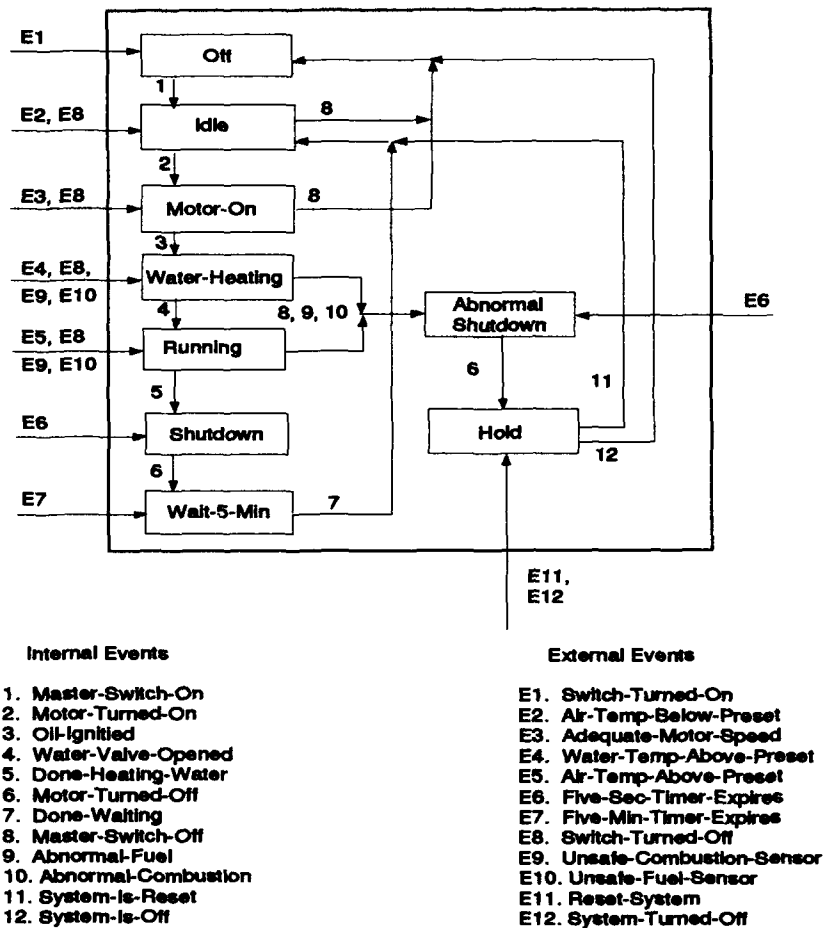


Figure 10. Home Heater State Transition Model

includes the instance-name and the attribute-name (e.g., *MOTOR.status*). Once the state objects have been created, Event objects, which model the transitions from state to state, should be created.

Events that cause a transition between two states in the system are modeled as internal events. There are two internal events which cause a transition out of the *MOTOR-ON* state. One event causes a transition into the *WATER-HEATING* state and the other event results in a transition to the *OFF* state. Event objects are very easy to create and we have named the first event, *OIL-IGNITED*. It is created as follows:

**OIL-IGNITED instance-of event**  
**type : internal**

The *type* attribute describes whether this event is from an internal source or an external source. External events do not cause a transition between two states, but rather indicate that certain external entities modeled by the system have changed value. To introduce changes to objects in the system, external events have an associated behavior. All of the Event objects must be associated with State objects, and Behavior objects (if applicable), in the Relation-Table.

In addition to its state-space attribute, a state is characterized by the activity being conducted in that state. Such activities, which are captured in Behavior objects, should be created next. The behavior associated with the *MOTOR-ON* state is defined by the *FURNACE-MOTOR-ON* object. This behavior is described by a decision table as follows:

<b>FURNACE-MOTOR-ON instance-of behavior</b>		
<i>MOTOR.speed</i> ,	<b>dont-care</b> ,	= adequate;
<i>MASTER-SWITCH.status</i> ,	= off,	= on;
-->		
<i>IGNITION.status</i> ,	off,	on;
<i>OIL-VALVE.status</i> ,	closed,	open;
<i>MOTOR.status</i> ,	off,	on;
<b>event</b> ,	<b>MASTER-SWITCH-OFF, OIL-IGNITED</b>	

Decision tables are broken down into four sections. The left-most column above the "-->" defines the pre-condition variables. The left-most column below the "-->" defines the post-condition variables. Each subsequent column stipulates one rule of the decision table. Each entry above the

"- ->" defines the required value for the pre-condition variable. Similarly, each entry below the " - ->" defines the value that must be assigned to the post-condition variable. For example, the first rule (second column) is read as: If the master-switch is turned off then turn the ignition off, close the OIL-VALVE, turn off the motor and then execute the MASTER-SWITCH-OFF event.

The dont-care symbol in the second column is an OML built-in value which is used to indicate that the variable defined in the left hand column does not affect the first rule. All column entries should be explicitly assigned some value or expression. This eliminates any potential ambiguity in the behavior specification. Therefore, if the value of a variable does not affect the rule, then the "dont-care" symbol should be entered. The Library problem will show how pre- and post-condition statements can be used to describe a behavior object.

Finally, the behavior objects must be associated with the state or event objects they describe. Since each state is characterized by an activity, a built-in association, ICO (is composed of), is provided to relate these two objects. The ICO association also applies to external events and their behaviors. Therefore, a State-Behavior association is entered into a Relation-Table as follows:

MOTOR-ON,      ICO,      FURNACE-MOTOR-ON;

These are the only types of objects and relationships needed to model the Heater Problem. This problem also shows the general process needed for translating ERMs and STMs into OML. The user's manual in Appendix F provides guidance on compiling, translating, and executing the OML specification. The next section presents a different type of problem. It requires a more static representation of objects that do not imply any sequencing of events.

*4.5.2 The Library Problem.* Unlike the Home Heater, the Library problem is not a state-based problem. This problem specifies the legal transactions that staff and ordinary users can perform, and the results of their actions. There are no sequencing requirements that stipulate an ordering of the operations. Therefore, this problem can be satisfactorily specified using an ERM

and a DFM. Since the translation of ERMs into an OML specification was discussed in the previous section, the translation of the Library problem's ERM into OML will not be discussed here. A complete description of the problem, its informal analysis, and the OML specification are provided in Appendix E.

This section describes the translation of a DFM into an OML specification. Data Flow Models typically do not provide any explicit notion of control structure. Without an explicit control structure, it is only possible to translate the DFM into a group of loosely coupled process objects with no particular execution sequence. That is, the individual processes are executable but there is no process sequencing information. Therefore, a function must be added to the REFINES executable specification which sequences and executes the processes. In the absence of control flow and control process information, the translation software will need to insert control information in the executable specification that locates a set of possible next processes and allows the user to select from this set. In the Library problem, no control sequencing was specified and therefore a group of loosely coupled objects will be created by the translation. A controlling function will be added to the automatically generated REFINES specification to sequence the processes that were defined in the specification and provide user interaction.

Appendix E contains all of the data flow diagrams necessary for analyzing this problem. The Process objects should be translated into OML first; however, not all processes identified in the DFDs will become process objects in OML. Only the leaf node processes need to be translated in OML. The leaf node processes are the processes at the lowest level of each numbered bubble in Figure 11. For example, processes 1, 2, and 3 can be further leveled into more detailed data flow diagrams and therefore are not represented into OML. However, if a process at this level existed in its most primitive stage (i.e. it could not be decomposed any further), then it would be translated into OML. Continuing with this example, the decomposition of process 2 is shown in Figure 12. Process 2 has been broken down into 11 distinct subprocesses. These processes are at their lowest



level of decomposition and each of these processes must be represented in the OML specification.

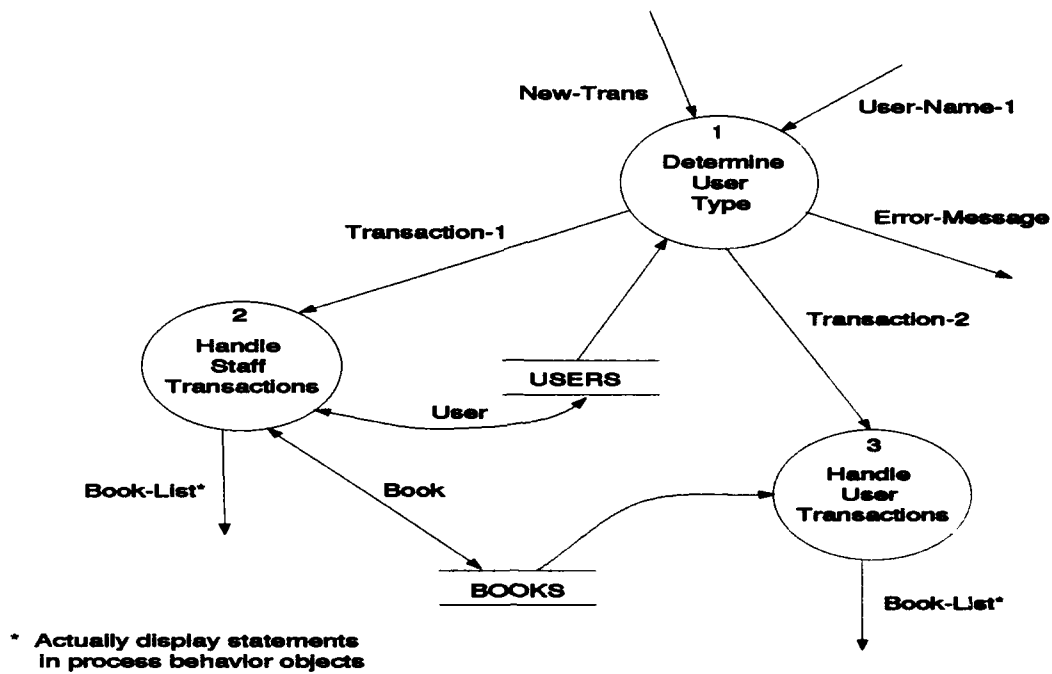


Figure 11. Library Problem Level 0 Data Flow Diagram

Using CHECK-OUT-BOOK (process 2.4) as an example, a process would be represented in OML as:

#### CHECK-OUT-BOOK instance-of Process

The declaration of the process object is not complete by itself. The behavior of the process now needs to be defined. The activity associated with the CHECK-OUT-BOOK process is captured in the CHECKING-BOOK-OUT behavior object.

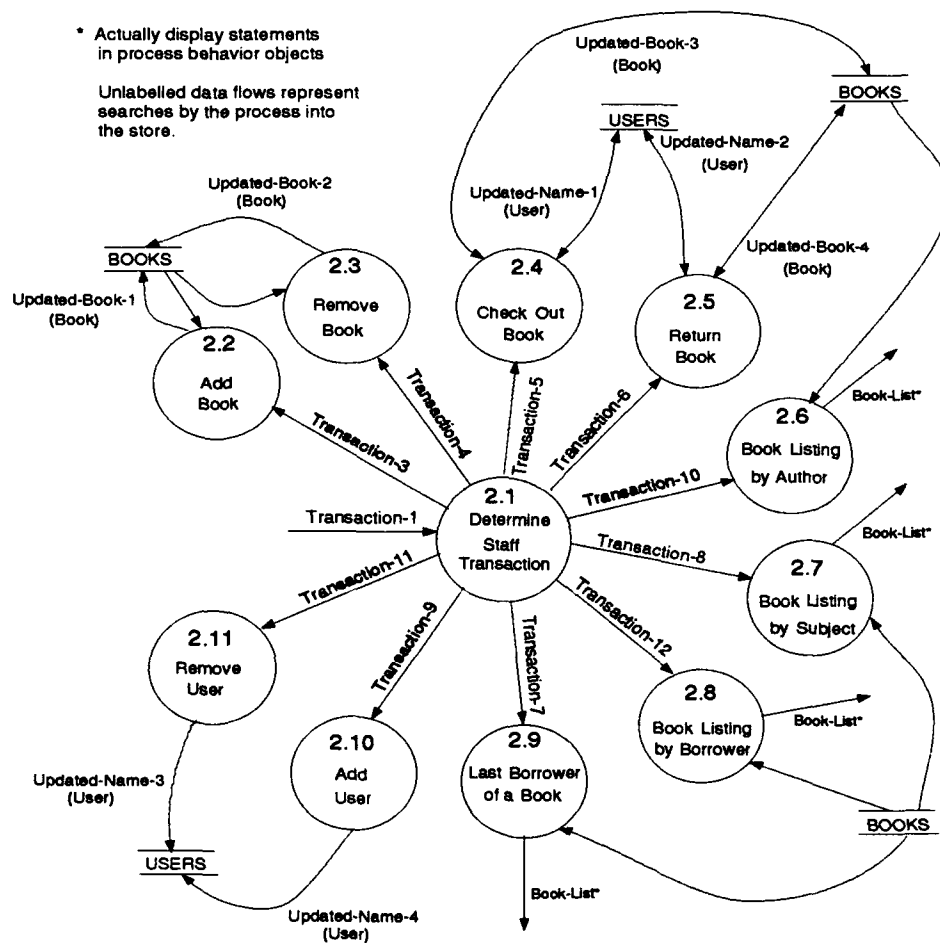


Figure 12. Library Problem Level 1 Data Flow Diagram

**CHECKING-BOOK-OUT instance-of Behavior**

```
exists (book) (book in BOOKS &
    book.book-id = TRANSACTION-5.Book-Id &
    book.status = available) &
exists (user) (user in USERS &
    user.user-name = TRANSACTION-5.Borrower-Name &
    user.book-count < 10)
- ->
UPDATED-BOOK-3 := getitem(book | book in BOOKS &
    book.book-id = TRANSACTION-5.Book-id) &
UPDATED-BOOK-3.status := checked-out &
UPDATED-BOOK-3.current-borrower := TRANSACTION-5.Borrower-Name &
UPDATED-NAME-1 := getitem(user | user in USERS &
    user.user-name = TRANSACTION-5.Borrower-Name) &
UPDATED-NAME-1.book-count := UPDATED-NAME-1.book-count + 1
event none
```

In this case, a Pre-Post-Condition statement was used to model the intended behavior. Notice the pre-condition and the post-condition are separated by the “- ->” symbol which indicates that if all the pre-condition requirements are true, then the post-condition statements should be made true. Additionally, the individual requirements of the pre-condition and post-condition are separated by ampersands. All pre-condition requirements are boolean expressions and all post-condition requirements are statements. Boolean expressions can be expressed by traditional predicate operators (e.g. <, >, = ...) or by the set operators, *in*, *forall*, and *exists*, which test if an element(s) is present in a set of elements. Postcondition statements can be either assignment statements, such as used above, or function calls. We found it necessary to provide a small set of built-in primitive object operations to supplement the user-defined processes. The built-in functions allow the user to create, destroy, and display objects. Example uses of these functions can be found in the complete OML specification for the Library problem (see Appendix E). Whatever is inside the parenthesis must be an existing entity or set of entities. Further, it is also important to note that expressions and statements in a process behavior refer to a flow object's attributes (flow-name.attribute-name). Preconditions test the current status of a flow's attributes, while post-conditions assign values to a flow's attributes.

Processes are also characterized by the data they operate on and the data they output. Therefore, Flow objects and Store objects must be created to represent the incoming and outgoing data. The Library problem contains a data store for all valid users and another data store for all valid books. The following is a representation of the stores in OML:

```
USERS instance-of Store
      nature : set
      content : User
BOOKS instance-of Store
      nature : set
      content : Book
```

The *nature* attribute indicates whether the ordering of the objects in the store is important. If ordering is important, then the *nature* attribute should be assigned the value *sequence*. The *content* attribute specifies the type of data the store contains. This value should be an entity class-name.

The CHECK-OUT-BOOK process is associated with several flows. One of the flows carries data from the CHECK-OUT-BOOK process to the data store, BOOKS. This flow represents the process object updating the BOOKS data store by sending it a Book object. This flow is defined as follows:

```
UPDATED-BOOK-3 instance-of Flow
      flow-link : proc-store
      flow-data : Book
```

The flow-link indicates the two types of objects that the flow object connects together. In this case, the data flows from a process to a store. The flow-data attribute characterizes the type of data that is carried by the flow object. The actual Book object that UPDATED-BOOK-3 carries is not determined until the REFINES specification is executed.

Once all of the processes, behaviors, flows, and stores have been defined, the last requirement is that they be associated together in a Relation-Table. This is accomplished in much the same manner as discussed for the Heater problem. Similar to specifying the relationship between a state and its behavior, processes are associated with behavior objects using the built-in ICO relationship. Additionally, flow objects are used to associate processes, stores, and entities.

#### *4.6 Summary*

The Object Modeling Language provides an intermediate format for translating an informal specification into an executable software specification. This chapter has presented the rationale for creating OML, the goals in defining OML's capabilities, the syntax and semantics of OML, an evaluation of OML's capabilities, and two test cases to informally validate the effectiveness of OML as a specification language.

The benefit of OML is clear. A problem specified in OML is now in a format which can be quickly and automatically translated into an executable specification which can be used to determine if the system requirements were properly captured in the specification. This capability enables the user to quickly uncover potential problems and ambiguities in the specification at a very early stage in the software development lifecycle which will save both time and money.

The next chapter will conclude our research by defining the process for converting OML specifications into executable REFINE source code.

## *V. Executable OML Specifications*

### *5.1 Introduction*

The objective of this portion of our research was to develop and implement a process for executing an OML specification. By observing the behavior of a specification, the specifier can determine if his specification accurately captures his requirements. As mentioned in Chapter III (see Figure 4), there are at least two ways to accomplish this objective. One way is to develop a translation process which converts an OML specification into REFINE source code that can be compiled and executed. The second method is to directly simulate the behavior of an OML specification. The first method is similar to the translation of a source program into an executable form, such as the compilation of an Ada program into executable object code. The second method is a direct simulation which executes the source program based on the internal representation of the parsed program. This method operates on an entire OML specification unit (i.e., process object, state object, behavior object, etc.), but does not explicitly perform a translation. The only output produced is the result of the behavior specified by the OML specification.

To simplify the process of translating OML specifications into an executable form, we translated the textual OML specification into an intermediate form – an Abstract Syntax Tree (AST). Since we chose to develop our software in the REFINE environment, we used DIALECT, REFINE's compiler-building tool, to construct a compiler that produces an AST. REFINE's environment contains many features for examining and manipulating ASTs. These features allowed us to more quickly and easily develop the translation software for generating an executable specification. REFINE's AST is useful to both behavior demonstration methods: compilable source code and simulation. The AST assists in the translation by allowing us to group like data items together and to associate objects that are loosely coupled in the OML specification. It also allows the translation software to work with information that resides in several different sections of the specification rather than requiring information to be operated on sequentially.

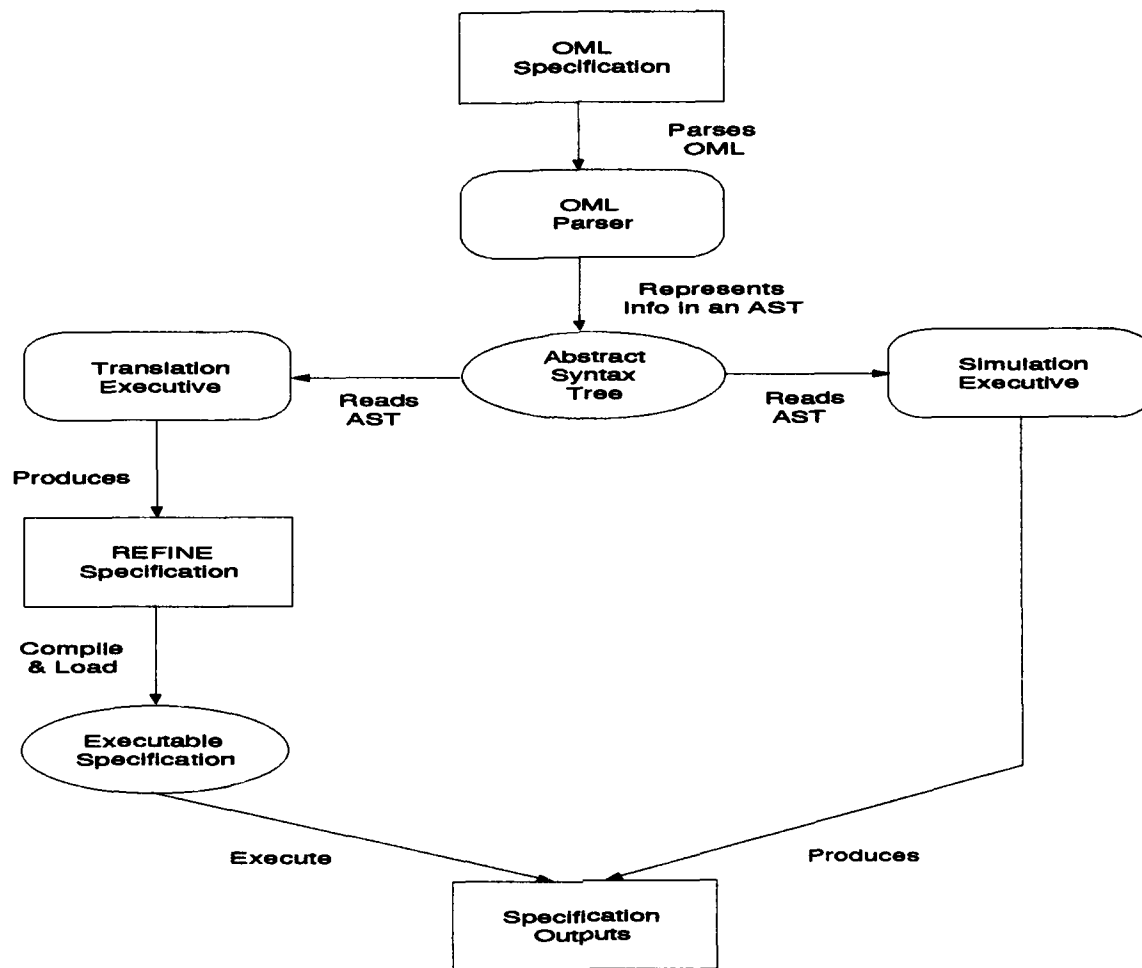


Figure 13. Steps Required for Translation versus Simulation

Figure 13 compares the steps required to prepare an OML specification for simulation versus the steps required to generate an executable REFINE specification.

1. The round-cornered rectangles depict executable programs that operate on other files to construct new products:

- The OML Parser transforms the textual OML specification into an Abstract Syntax Tree (AST).
- The Translation Executive, or translation software, operates on the AST and produces a file that contains a compilable REFINE program that represents the parsed specification.

- The Simulation Executive, yet to be developed, is a program that traverses the AST and displays outputs on the terminal as though the specification were executing.
2. The rectangles depict files that are processed by the executable programs:
- The OML Specification is produced by an elicitation tool (recommended method) or manually (until the elicitation tool is developed) and represents a system that has been analyzed and decomposed using informal software modeling techniques.
  - The REFINE Specification is an automatically generated, compilable REFINE program.
3. The ovals depict the machine representation of files that exist as objects in the REFINE environment (object-base).
- The Abstract Syntax Tree (AST) is produced when an OML specification file is parsed into the REFINE environment.
  - The Executable Specification is generated when the REFINE Specification is compiled and loaded into the REFINE environment.

Due to time constraints, we focused on the translation of OML specifications into executable REFINE programs. As part of a previous project, we wrote a compiler that parses an Ada program into a REFINE AST and translates the information in the AST into REFINE source code. The result was a REFINE program that displayed behavior identical to the Ada program. Because we understood how to perform the translation from OML to REFINE, we were able to focus on other problems associated with the representation and interaction of informal model objects needed to develop an executable specification. Also, because our goal was to create an executable specification, it appeared more straightforward to associate informal model objects with REFINE program components than to pursue the more abstract approach of behavior simulation directly from the AST. This aspect is left for future exploration. The simulation approach may be more appropriate as the size of the specification grows larger. Once a change or correction is made to the specification, the specification



would only require re-parsing to simulate behavior, rather than re-parsing, re-translation, and re-compilation as is needed to generate executable REFINE code.

Several steps are required to transform an OML specification into an executable form that can be used to verify correct behavior. All of these steps are performed by the translation software which the user can initiate by entering one command and each step is outlined below. (See Appendix F for a more detailed explanation of the steps involved in the automatic translation.)

1. First, OML specification is converted into an intermediate form, a REFINE AST, by parsing the specification through a compiler generated using DIALECT. Section 5.2 contains a detailed discussion of how this was done.
2. Second, the information in the AST is translated into a REFINE program that can be compiled and executed.
3. Last, the specification is compiled and loaded into REFINE's environment and the user is prompted to execute the specification to demonstrate its behavior. Testing the specification is not entirely automatic. It requires the user to select desired paths through the executable specification to simulate external inputs to the modeled system.

This chapter discusses compiler generation using DIALECT, the generation of REFINE source code from the AST, the process required to generate executable specifications from state-based models and process-based models, and the REFINE construct that each object in the Unified Abstract Model (UAM) is translated into.

## *5.2 OML Compiler Generation*

A compiler was needed to transform textual OML specifications into an intermediate form suitable for automated translation. Using the Backus-Naur Format description of OML's grammar in Appendix A, we generated a compiler using DIALECT, REFINE's formal language manipulation tool. In order to build a compiler, DIALECT requires that a domain model and a grammar be

constructed. The domain model describes the types of objects that make up OML. (Objects can be thought of as nodes on a directed graph.) These objects are modeled as classes and subclasses of one another to form an “is a” hierarchy. Using this hierarchy, the grammar can replace a “parent” type with one of its subtypes when parsing an input stream. Objects can also be paired together using REFINE maps. Object attributes are created by pairing one object with another object. Defining an object’s attributes is equivalent to building an “is composed of” relationship. (Maps equate to the arcs in a directed graph.) A grammar defines for DIALECT how the objects in the domain model are composed into a language. That is, the grammar identifies how objects and punctuation can be grouped together into sentences. As the compiler parses a file, it identifies objects in the domain, and builds a parse tree based on the maps defined in the domain model and the rules (productions) contained in the grammar. Diagrams of the OML domain model are found in Appendix C. We used the standard lexical analyzer provided with DIALECT to parse the input. OML did not require us to customize the parser or build our own AST-building routines.

We felt that incorporating an Ada-based Program Design Language (PDL) into OML would give the specifier more flexibility in describing behaviors. A compiler for a subset of Ada that could serve as a PDL standard was written as a compiler class project. In addition to converting an Ada program into an AST, the Ada PDL compiler also performs semantic checking. However, merging one grammar into another was not a simple task. The OML and Ada PDL domain models and grammars were compared to determine what common structures had to be developed. Common structures were required because both Ada PDL and OML use the same mathematical expressions. DIALECT has a mechanism for disambiguating a grammar by specifying the precedence of operators in expressions. However, the precedence definition can only apply to one REFINE object class. Therefore, all the mathematical expressions for both Ada PDL and OML had to be represented with a single parent object class. All other objects in the domain models were made unique to either Ada PDL or OML. The Ada PDL was written to be stand-alone. OML was then structured

to inherit all the grammar productions (rules) from the Ada PDL grammar <sup>1</sup>. Care had to be taken when assigning names to objects and maps in both the Ada PDL and the OML domain models to ensure that nothing was redefined or omitted. If Ada PDL and OML both have an object class or a map in their domain models named *X*, REFINE will use the definition of *X* that is compiled and loaded last. That is, it will over-write any existing definition with the same name. Should this occur in the definition of a map, the “is composed of” relationship created could be quite different from the one that should have been made. Similarly, if like-named objects have been placed in both the Ada PDL and OML domain models, productions for these objects should only be defined in one of the languages’ grammar. If Ada PDL and OML both define a production (a grammar rule) for the same object, only OML’s production will be used. This is because DIALECT will only copy productions from an *inherited* grammar (Ada PDL) that do not exist in the *inheriting* grammar (OML).

The two grammars were successfully merged, but there was insufficient time remaining to update the Ada PDL semantic-checking and translation software to reflect the extensive changes made to the names in the Ada PDL domain model. Semantic checks need to be created to ensure that attribute names are properly converted to the correct REFINE syntactic form, and that branching (outside of the specified behavior) is prohibited. Further research is required to determine how constraints are applicable to behaviors specified using PDL, and if appropriate, how they should be applied. A detailed discussion of constraints can be found in Section 5.4.10. The domain models and grammars can be found in Appendix B, Sections B.1 and B.2, respectively. The OML domain model and grammar that do not incorporate Ada PDL are located in Appendix A, Section A.3 and

---

<sup>1</sup>This may seem odd at first, because it implies that Ada PDL is at a higher level than OML. But we did so because Ada PDL contains a more extensive description of expression objects. DIALECT was designed to allow multiple dialects of a language. The most general case of the language is created with its own grammar. Each variant of this language can be created by building a grammar that has productions expressing only the variations needed for the dialect and inheriting the rest of the language’s productions from the general case. The *inherits-from* command causes DIALECT to copy only the productions from the inherited grammar with left-hand-sides not already defined in the inheriting grammar. The variant language’s grammar must also include any precedences, start-classes, or other information that is necessary for the grammar, because these things are not inherited from the general case. Because OML objects were the highest level objects in our model, and because lower level expression objects were shared to allow the precedence rules to disambiguate both the Ada PDL and OML grammars, Ada PDL was made the generalized case.

Section A.4, respectively. For further information on using DIALECT to generate a compiler, see (21).

### *5.3 Executable Specification Methodology*

Before describing how each OML object is translated into REFINE, we will describe the methodology used to compose the OML objects into an executable REFINE specification, that is, a REFINE program. This should give the reader a better understanding of how all the OML objects interact with each other to produce an executable specification. Also, this should allow the reader to understand why each object was translated in the chosen manner.

OML specifications are translated into a group of REFINE data definitions and functions that represent the objects in the specification. These objects, however, do not create an executable specification, because no top-level control module is specified in a DFM or an STM. Therefore, the translation software also inserts one of two different controlling functions into the executable specification. The controlling function for state-based models directs the executable specification's program flow based on events that occur. The controlling function for process-based models selects the next process to execute, based on which data flow objects have valid information in them. The translation software determines which function to insert into the executable specification based on the types of objects in the OML specification. Currently, a controlling function does not exist for specifications that contain both process- and state-based models simultaneously. The integration of both of these models into the same executable specification needs to be developed in future research. Regardless of which controlling function is added, the controlling function is named "sim" and the specification is executed by typing "(sim)" at the REFINE prompt. First, the execution of state-based models will be discussed.

*5.3.1 State-Based Model Execution.* State-based models are typically composed of entity objects, state objects, event objects, and behavior objects (as well as relation tables). As alluded to

in Table 4, state and behavior objects are converted into functions, entities are converted into object classes and instances, and events are used to develop the sequencing of actions in the executable specification. A controlling function is then added to sequence the execution of the state and

OML Object	REFINE Construct
Entity Class	Object Class and Variable Maps
Entity Instance	Object Class and Object Instance
Relationship	Not currently used or translated
State	Function
External Event	Symbols
Internal Event	Used during behavior translation to locate the next state, but not directly translated
Behavior	Function
Process	Function
Flow	Object Instance
Store	Variable (Representing a Set of Objects)
Relation Table	Used during translation to associate related objects, but not directly translated

Table 4. Mapping OML Objects into REFINE Executable

behavior functions to produce an executable specification. Figure 14 illustrates the flow of control as “sim” executes.

Upon translation, the name of the system’s initial state is written into the controlling function. The initial state is assumed to be the first state declared in the OML specification. The controlling function first makes a call to the initial state function. The initial state function, as well as all state functions, perform the following operations:

- It tests the system’s current state-space against the state-space required for that state as defined in the OML specification.

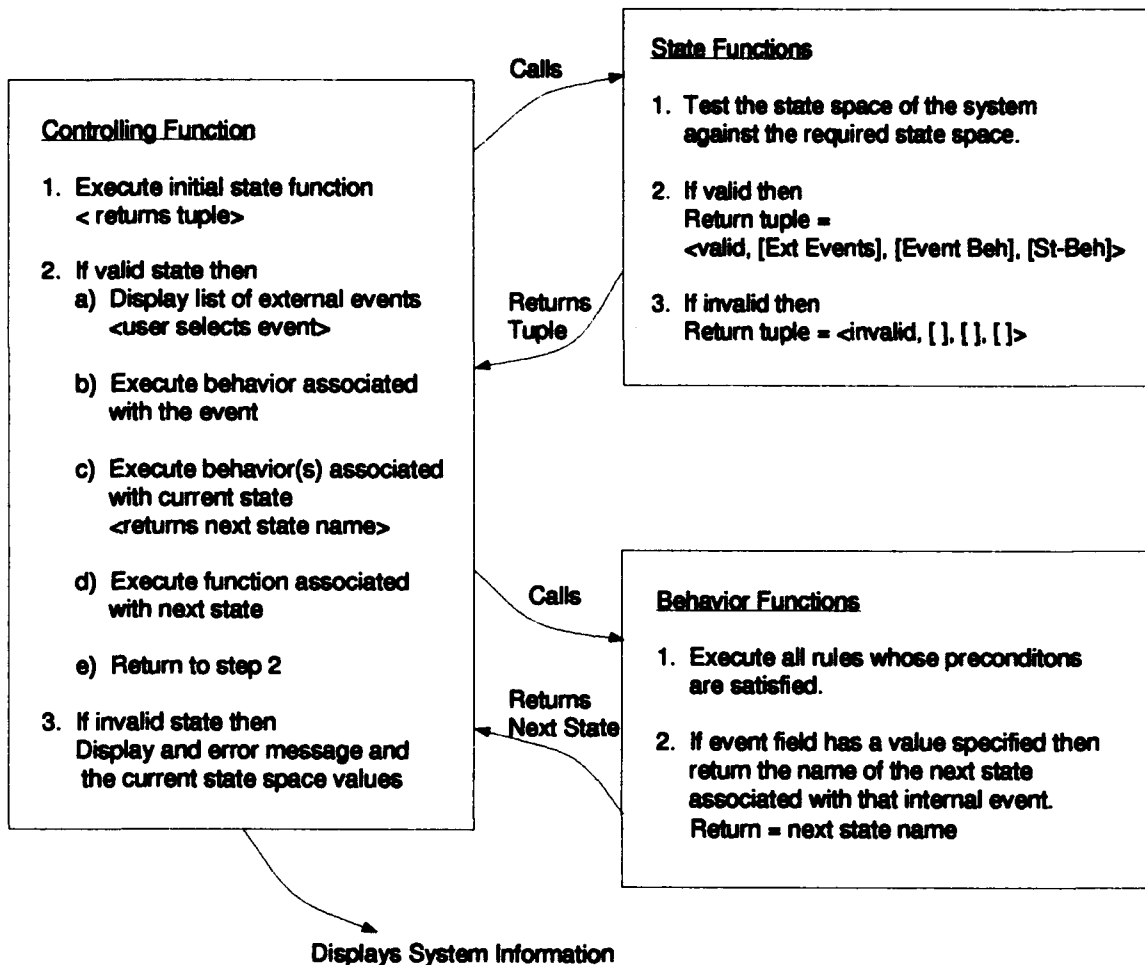


Figure 14. State Based Model Execution Methodology

- If the state-space is valid, the function returns a tuple of information to the controlling function. The tuple contains:

- a valid state-space flag,
- a sequence of external events that can occur during that state,
- a sequence of behaviors corresponding to the sequence of events, and
- a sequence of behaviors that characterize the current state.

This tuple is used by the controlling function to direct the sequencing of actions.

- If the state-space is invalid, the tuple is assigned an invalid flag and three empty sequences, and the tuple is returned to the controlling function.

If the tuple returned from the state function indicates the state-space is valid, the controlling function next displays a list of external events that the user can choose from. By selecting an external event, the executable specification is able to simulate the introduction of external events into the system. Currently, handling multiple events and ranking events by priority is not supported, but it could be added with little difficulty. The behavior associated with the selected event is then executed, which modifies the value of specified objects.

The controlling function next executes the behavior functions associated with the current state. These behaviors perform the same functions as external event behaviors; however, these behavior objects return the name of the next state. If the event field of the behavior object is specified, then the behavior function returns the name of the next state to the controlling function. The translation software uses the event name to search through the relation table to find the name of the next state associated with the event. As a rule, external event behaviors cannot provide a value in the event field since they only cause changes to state-space values. Therefore, no information is returned from an event behavior function. The name of the next state is used to restart the same cycle of events described above.

If at any time the system enters an invalid state, the executable specification lists the current values of each object in the system and prompts the user to compare those values against the values required for that state by the OML specification. This is very helpful in uncovering inconsistencies and incompleteness in the user's informal specification.

**5.3.2 Process-Based Model Execution.** Automatically generating an executable specification from process-oriented models proved to be more difficult than state-based models. The two main reasons for this were the representation of flow information and the lack of control sequencing information. The first problem was how to represent data flows in REFINE. We needed a naming

convention to allow us to associate specific data on a flow with the data items manipulated by a process' behavior and the data items contained in a store. We had two choices: to model the flows implicitly by passing parameters between functions or to model them explicitly by declaring each flow as a global variable. We chose to implement flows as global variables. This provided a fixed set of names for the behaviors to reference and eliminated the need to parameterize the function calls from processes to their respective behaviors. Further discussion of the pros and cons of these two options is provided in Section 5.4.7. The second difficulty encountered was determining how to develop an executable specification from an OML specification that does not contain any control flow information. Lack of control information requires increased user interaction with the simulation software. This discussion does not address how control process and control flows should be integrated into the executable specification. This functionality must be addressed in future research to make the translation software more robust.

Process models (DFMs) represented in OML are composed of process, flow, store, entity, behavior (describing the process' actions), and relation table objects. Table 4, page 92, defined how each of these types of objects are represented in the executable specification. A controlling function is added during translation to compose these objects into an executable specification. Figure 15 illustrates the flow of control during execution. The controlling function begins by displaying a list of processes to the user. The user selects one of these processes as the initial process which the controller executes. Process objects are translated to perform two operations: Check to see if all its internal in-flows are satisfied, or perform transformations on its in-flows and generate out-flows. The controlling function passes a parameter (either "check" or "execute") to the process function to indicate which operation the process function should perform.

If the controlling function selects "check", the process determines if all the in-flows from internal sources have been provided to the process. If any of them are not defined, then the



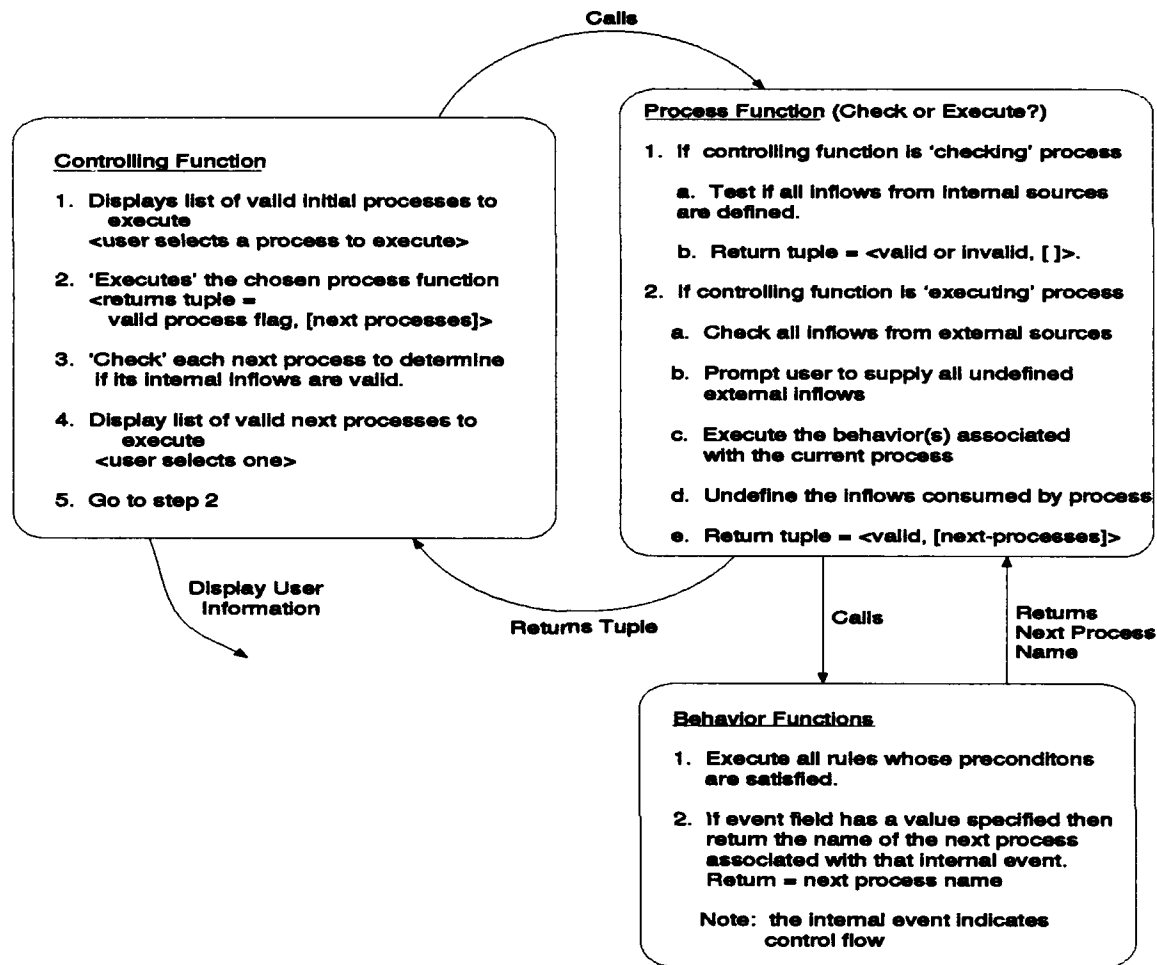


Figure 15. Process Oriented Model Execution Methodology

process will not have all the information it requires to execute and an invalid flag is returned to the controlling function. This indicates the process is not eligible for execution<sup>2</sup>.

If the controlling function selects "execute", then the following operations are performed:

- If any required in-flow from an external source is not available, the process function prompts the user to enter the required data.

<sup>2</sup>This may seem to constitute a restrictive user interface. Our goal was to display a limited set of next available processes for the user to select from during execution of the specification. We did this by locating the processes whose internal in-flows were valid. This resulted in a specification that executes in the order that the flow-data is generated. The execution would be more flexible if any process could be selected for execution at any time. If a process does not have all its in-flows satisfied (whether internal or external), then the software would prompt the user for the information. This feature would require the process-based controlling function to be restructured in future research.

- Once all in-flows are satisfied, the behavior function associated with the process is executed. Behavior functions perform the same operations as mentioned in the state-based model execution, except that process behaviors cannot cause any type of branching or calling to another behavior function unless control information is specified.
- After the in-flows to the process have been consumed, the data carried by the in-flows becomes un-defined.
- The process returns a tuple containing a valid process flag and either the name of the next process to execute (if control information is provided in the process' behavior) or a list of potential processes to execute based on the current state of the simulation.

The controlling function then repeats the same cycle of events over again by displaying the list of processes that can be executed next. The next several sections provide details on how each OML object is converted into REFINE or used by the translation software.

#### 5.4 Translation

Similar to the tasks we performed when developing the UAM, we needed to evaluate the major components of an executable program and ensure that the translation software built an executable specification that correctly represented the OML specification. To develop an executable specification, we needed to determine three major groups of information from the AST representation of the OML specification:

- the data architecture (data that needs to be manipulated),
- the behavioral architecture (manipulations to perform on the data), and
- the control architecture (the order in which manipulations occur).

The data architecture is determined from the OML entities, stores, and flows. These are passive objects that describe the data or objects the specification works with. The behavioral architecture

is described by behaviors, processes, and states. These objects specify the operations that must be performed on the data. The control architecture is derived by the translation software from information found in events and flows. The translation software adds sequencing information to each state and process that is used by a controlling function to order the execution of state and process functions during the simulation. Depending on whether the OML specification contains state-object descriptions or process-object descriptions, one of two different controlling functions are added into the executable REFINE specification. Both controlling functions require user interaction to direct certain steps during the execution of the specification. That is, a state-based model requires the user to select the next external event from a set of external events to cause a state-space change. Similarly, process-oriented models require the user to select the process to execute next.

Table 4, page 92, summarized the general mapping of OML objects into executable REFINE code. These REFINE objects and functions are composed together with a controlling function for either a state-based or transaction-oriented model.

The correctness of the translation process was informally validated through the execution of the two test cases: The Home Heater and the Library System problems. The results of their execution clearly modeled the intended behavior of these systems. While it is not appropriate to conclude that the translation process is totally correct, we have achieved a measure of success. Both problems were successfully translated from their OML specifications into their associated executable specifications. Testing showed that the executable specifications actually captured the behavior as specified by their OML specifications. Care was taken to form general behavior translation routines so that the translation software was not biased by our knowledge of the expected behaviors of the two sample problems. In each case, we intentionally injected a controlling function that was not present in the OML specification. This function directs the execution of the specification based on selections made by the user. Therefore, based on the successful generation of executable

specifications for these two problems, we can say that our translation process is correct for the two instances in which it was tested. Our translation software did not convert constraints, relationship objects, and Program Design Language behavior objects into the executable specification. In this sense, the translation software is not complete. Hence, it is not fully validated. The process has been validated, however, for entity objects, event objects, Decision-Table and Pre- and Post-Condition behavior objects, flow objects, store objects, state objects, and process objects. The inclusion of constraints and relationship objects in the translation will impact the behavior of the executable specification requiring the translation to be re-validated.

OML can also support a more formal proof of correctness of the translation process. Two other methods can be employed to perform a more extensive validation of the translation process. Validation can be performed by establishing an extensive set of test cases to test the translation's correctness and completeness. Because of OML's basis in first-order predicate logic, theorem-proving is another validation technique that can be used. Future research will address these important issues.

The details of each object's translation are discussed in the following sections.

*5.4.1 Entities.* The Object Modeling Language allows the definition of both entity class objects and entity instance objects in an OML specification. Both of these OML constructs are used to define the data architecture of the system.

In the first case, entity classes are translated into their REFINe equivalent: object classes. Additionally, the attributes associated with these entity class objects are translated into REFINe attributes by declaring a REFINe variable that maps the object class to the data type of the attribute. For example, consider the following entity class definition from the Home Heater specification (see Appendix D):

```

VALVE class-of entity
  type : external
  parts
    status :symbol range {open, closed}

```

This entity class definition gets translated into the following REFINE constructs:

```

var VALVE : object-class subtype-of HOME-HEATER
var VALVE-STATUS : map (VALVE, symbol) = {||}

```

The mapping of entity class objects into REFINE is relatively straightforward; however, there are a few translation issues that need further explanation. Every entity class object has a required *type* attribute that declares whether the entity is external or internal to the system. This type attribute is not directly translated into REFINE code but is used by the translation software to determine if a data flow comes from an internal or external source.

The REFINE attribute names are a concatenation of the object class name and the entity attribute name. This convention was adopted to ensure that all attributes defined in the translated code have unique names. If the attribute names are not unique, then only the last attribute compiled by REFINE would be defined. All previous attribute definitions would be overwritten.

The VALVE object-class is a subtype-of HOME-HEATER. The name of the top level REFINE object in any translated file is defined by the name of the specification. That is, the name following the keyword *specification* at the top of the OML file becomes the top level object in the REFINE executable specification. Therefore, all object-classes in the translated file are defined to be subtypes of the top level object. This feature provides a convenient method to locate and manipulate all the data objects associated with any given REFINE specification.

The range of legal attribute values is currently not translated into REFINE. Currently, this information is not used but should be used to do static semantic checking on the values assigned to the object attribute. Due to time constraints, code for semantic checking was not developed. At least two options exist for performing the semantic checking. First, the semantic checking can be performed by the elicitation tool that generates OML specifications to ensure that only valid

OML specifications are provided to the translation software. The second option is to implement the semantic checking as part of the translation process. Future research will address these issues.

The second type of OML entity objects are entity instances. Two types of entity instance objects can be defined:

1. *Instances of a user-defined class.* The translation of instances of a user-defined class is a three step process:
  - (a) Declare a variable of the object class type.
  - (b) Make an object of the object class type and assign it to the declared variable. This is accomplished by using REFINE's *make-object* command.
  - (c) Define the attributes of the declared variable. This is done by using the *set-attrs* command.

All three steps can be included in one REFINE statement as shown below. The OIL-VALVE entity instance from the Home Heater gets translated into the following REFINE code:

```
var OIL-VALVE : VALVE =  
    set-attrs(make-object('VALVE),  
                'name, '*OIL-VALVE,  
                'VALVE-STATUS, 'CLOSED)
```

The name attribute of the object is a REFINE built-in attribute and is needed to use several of REFINE's object manipulation functions. The name assigned to each object also must be unique to other object names as well as unique from the name of the object's variable declaration. Therefore, we defined each object's name attribute to be the variable name preceded by an asterisk (\*). This allows REFINE to distinguish between the actual object and the name of the variable declaration.

2. *Instances of OML's built-in entity class.* This translation process is a combination of defining an entity class and defining its instance. First, an object class is defined for each instance of the built-in entity class, then an object is created to represent the entity instance. The following is an example of this translation:

As defined in the OML specification:

```
MASTER-SWITCH instance-of entity
  type : external
  parts
    status : symbol range on, off init-val off
```

The translated REFINE source code consists of the class declaration:

```
var MASTER-SWITCH-ENTITY : object-class subtype-of HOME-HEATER
var MASTER-SWITCH-ENTITY-STATUS:
  map(MASTER-SWITCH-ENTITY, symbol) = {||}
```

followed by its instance declaration:

```
var MASTER-SWITCH : MASTER-SWITCH-ENTITY =
  set-attrs(make-object('MASTER-SWITCH-ENTITY),
    'name, '*MASTER-SWITCH,
    'MASTER-SWITCH-ENTITY-STATUS, 'OFF)
```

To keep the object class name unique, we chose to define the object class name as the concatenation of the entity instance name with the word ENTITY.

*5.4.2 Relationships.* An Entity Relationship Model (ERM) illustrates the relationships between entities in a system. OML allows the creation of three types of Association objects: Relationships, Events, and Flows (see page 46). Relationship objects model the static relationships between entities found in ERMs. We created these OML objects under the assumption that *all* the objects contained in the informal models would be useful in OML. However, when developing our translation software to create an executable specification, we did *not* find a need for Relationship object information<sup>3</sup>. We were able to develop what we believe is a generalized translation process that does not use the information provided by Relationship objects. This is not to say that Relationship objects are useless to OML. The two problems we used to test our system, the Home Heater and Library problems, are limited in size and complexity. The implementation of more difficult modeling problems in OML may display the applicability of Relationship objects for generating an executable specification. It is not surprising that Relationship objects are not used in

---

<sup>3</sup>This lack of usefulness was not the case with other association objects, or the built-in ICO relationship that occurs in the Relation Table. The dynamic nature of Flows and Events makes this information indispensable in creating an executable specification. The Relation Table lists all the user-defined Associations and the compositions (ICOs) of State and Process objects with their Behavior objects. The table was also used extensively in the translation process to locate objects that originally were referential attributes in the original OML architecture (see page 41).

generating an executable (dynamic) specification. Relationship objects represent the relationships in ERMs. These relationships describe the data structure of a system. Developing an executable specification requires the modeling of the dynamic relationships and behavior of a system. REFINe provides object definition and management functions. That is, we did not develop our own data base to store and catalog objects. If we had, we may have found Relationship objects useful in structuring the data base. For the present time, it appears reasonable that Relationship objects do not affect the generation of an executable specification.

As a point for future research, Relationship objects may be useful for defining constraints on the operations that one class of objects may perform on another class of objects. For example, Figure 25, page 276, shows the ERM for the Library problem. A staff user can *add/remove* a Book from the Library. If the Add relationship object was associated in the relation table with the Adding Book behavior, then the Add relationship object could be used to constrain who is allowed to add a book to the Library. In our analysis of the Library problem, the constraint that only a staff user can add a book to the library was built into the pre-condition of the Adding Book behavior object by using the *in* set-expression operator to see if the user was in the set of staff users.

**5.4.3 States.** The translation software converts each OML State object into a REFINe function that performs the operations as described in Section 5.3.1, illustrated in Figure 14 (page 93).

In order to generate a REFINe function that accomplishes these tasks, the translation software performs the following actions for each state object:

- By gathering information from the relation table, the translation software creates a sequence of external event names applicable to the current state. It declares a local variable in the REFINe state function, and assigns the event sequence to the variable.



- At the same time it creates a sequence of behaviors that correspond with the external event sequence and assigns this sequence to another local variable in the REFINE state function.
- It generates a sequence of behavior object names associated with the current state and assigns that sequence to a third local variable in the REFINE state function.
- It uses the state-space attribute defined for each state to create an if-then statement in the function to test the system's state-space and return the appropriate tuple as mentioned above.

Two notable problems surfaced during this translation. First, the state-space attribute references needed to be translated from "dot" notation to a form acceptable to REFINE. That is, object attributes are written in the form of *object-name.attribute-name*. When referencing object attributes, state-space constraints model the arguments of their expressions using "dot" notation. This presents two problems. First, this is not the same notation that REFINE requires. REFINE refers to an object's attribute as *attribute-name(object-name)*. Second, the translation software converts each object's attribute name into a unique attribute name. As a result, when converting state-space constraints, the translation software must find the unique name of the attribute in the attribute table (created when Entity objects were converted into REFINE, see Section 5.4.1), and also correctly format the syntax for finding the attribute value. The second problem deals with the semantic checking of the state-space attribute. The state-space attribute defines the required values of all object attributes important to that state. A semantic check must be added to verify that the values of the attributes, required by the state-space attribute, are within the legal range of values specified by the entity object. This semantic checking can be performed by the elicitation tool that generates the OML specification or by future modifications to the translation software depending on whether static or dynamic constraint checking is desired.

**5.4.4 Events.** Events are critical to developing the control architecture of an executable specification. OML allows the creation of both internal and external events, each of which is treated differently by the translation software.

Internal events in OML are used to link one state of the system to another state and do not possess any behavior. They are referenced in the event field of Behavior object rules to indicate which state the system should transition to next when the behavior rule is executed. Therefore, whenever an internal event object is referenced in a behavior object, the translation software uses the relation table to find the name of the next state associated with that event. The discussion of Behavior object translation will describe how the next state name is used.

External events differ from internal events in that they possess behavior which is used to change the state-space of the system. They are not used to link two states of a system together, but rather are used to specify a change in the values of state-space variables which in turn may allow one or more activities to occur in the current state. (The activities of a state are specified in the behavior object associated with the state). External events represent actions of objects that exist externally to the system. Since these events need to come from external sources, this presented a problem in developing an executable simulation. To satisfy this need, we added a user interaction routine to the translated executable specification which interacts with the user each time the system changes state. The interactive routine displays a list of external events that the user can select for execution while in that state. Once the user selects an external event, the behavior associated with that event is executed. The event's behavior is responsible for changing the state-space variables.

*5.4.5 Behaviors.* Behavior objects perform an integral part in defining the behavioral architecture of an executable specification. Recall that Behavior objects in OML can be defined by Decision Tables (DT), Pre-Post Conditions (PPC), and Ada PDL. Each of these types of Behavior objects are represented very differently in OML, therefore each requires its own unique translation process. However, the representation of both DTs and PPCs in an executable REFINE specification are very similar, therefore the translation software performs the same types of operations on these objects. In both cases, each PPC rule and each column in a DT is converted in the following manner:

- the pre-conditions of the rule, or the conditions of a decision table column, are located in the AST and translated into REFINE,
- the post-conditions of the rule, or the actions of the DT column are translated and
- the rule's next event (if specified), or the column's next-event (if specified) is translated.

However, DTs and PPCs are converted into different REFINE constructs. Each DT Behavior object is transformed into a REFINE function that consists of one REFINE "transform" construct for each DT rule. Similarly, each PPC behavior object is converted into a REFINE function, but instead consists of one "if-then" construct for each pre-post-condition rule. The rationale for translating DTs and PPCs into different REFINE constructs is provided later in this section during the discussion of problems encountered.

PDL Behavior objects are translated into a self-contained REFINE program. The translation code for converting Ada PDL into REFINE is provided in Appendix C. However, the PDL translation and semantic checking code (trans-pdl.re, sem.re, and tcheck.re) must be updated to reflect the name changes made to PDL's domain model and grammar which were necessary to enable the PDL and OML grammars to operate concurrently. The required modifications should be straightforward; however, time constraints did not allow us to accomplish this task.

The REFINE functions that represent OML behaviors also contain information that helps the controlling function order the execution of the state and behavior functions. The event information in PPCs and DTs is optional. It allows the user to specify an internal event that will occur if that rule or column in the behavior is executed. The translation software uses the name specified for the event and searches the relation table to find the name of the next state associated with that event. The name of this next state is then returned by the behavior function to the calling function. If an event is not specified, then nothing will be returned. This indicates that user interaction will be required for internal event selection as well as external events. State-based models should provide

an event for each rule to link the current state of the system to the desired next state. Process based models will only provide events if control flows are specified in the system's DFM.

*5.4.5.1 Problems Encountered.* When developing the behavior translation code, the following difficulties had to be resolved:

- When referencing object attributes, behavior rules model the arguments of their expressions and statements using the "dot" notation. Because REFINe does not access object attributes as *object-name.attribute-name*, the names are converted to *attribute-name(object-name)* by the translation software. This is accomplished using a look-up table that is created as entity objects are translated and given unique names.
- Originally, we designed the translation software to produce one REFINe "transform" construct for each rule. We preferred to use transform constructs since they are better suited for specifying *what* as opposed to *how*. A REFINe transform is structured as a pre-condition predicate and a post-condition predicate separated by a transform symbol (e.g.,  $P \longrightarrow Q$ ). It is interpreted as meaning: if the initial state-space is P then make the final state-space Q. It is similar to an "if-then" construct *except* that it does not explicitly state *how* to attain the final state. In that same vein, the order of execution and the manner in which the post-conditions are satisfied is determined at run-time by the REFINe compiler. Therefore, it is more a specification of *what* to accomplish as opposed to an implementation procedure. Since the order of execution cannot be determined by the user, all post-condition statements must be independent of each other. Unfortunately, when testing the executable Library specification, the order in which the post-conditions were executed was important for certain Library functions. This seemingly non-deterministic behavior resulted in a run-time error which was a direct result of the order in which the post-conditions were executed. To resolve this problem, we modified the conversion of PPCs to generate one "if-then" construct for each PPC rule.

This problem also spurred further thought. Perhaps writing a PPC behavior specification in OML that requires a specific order of execution is an abuse of the intent of a pre-post-condition(22:3-159). PDL is a better choice for these kinds of behaviors. This being the case, we submit that once PDL becomes fully integrated with OML, the translation of PPCs into REFINE must revert back to using the transform construct.

*5.4.6 Processes.* Data Flow Models use processes to represent the transformation of data or the actions performed on data in a system. A process in an informal model can be described by the data it uses (an input flow), the data it produces (an output flow), and the activity (behavior) of the process. The number of behaviors needed to describe the activity of a process depends on the desired level of abstraction. DFM's provide an excellent overview of the important functional components of a system but do not provide any details on the transformation of the data. These are contained in the process specifications (p-specs) that accompany the models (29:68).

Similarly, OML process objects also represent data transformation functions and are translated into REFINE functions. However, OML process objects do not possess attributes to directly model the data that they consume and produce, or their behavior. Instead, the relation table is used to associate each process with its incoming data flows, its outgoing data flows, and its behaviors. The description of the transformations performed by the process are contained in its behavior specifications.

Figure 15 (page 96) illustrates the operations that each process function in the executable specification must perform. Each process is responsible for performing one of two different operations. The controlling function specifies which operation the process should perform by passing a parameter to the process when it is called. One of a process' operations is to ensure that its internal in-flows are defined and to return a valid/invalid flag to the controlling function. The other operation requires the process to execute its behaviors, un-define each in-flow to the process after it has been consumed (used) by the behavior, and return a list of processes that can be ex-

cuted next to the controlling function. To support these operations, the translation software builds into each process function several pieces of information. Each process maintains two sequences of in-flow object names. One sequence contains the names of all in-flows to the process that come from internal sources. The second sequence contains the names of all in-flows that originate from external sources. These sequences are used by the process function to verify that its in-flows have valid data. If any internally generated flows do not have data in them, an invalid flag is set in the return-tuple and the tuple is passed back to the controlling function. For a more detailed discussion of the interaction between the process function and the control architecture see Section 5.3.2 (page 94).

However, before we could develop an automated process for translating OML processes into REFINE functions that perform these operations, the following issues had to be resolved:

1. How to incorporate into the process function the flow objects that are pertinent to it,
2. How to find the processes that can be executed next,
3. How to validate the data in a flow object when the process is executed,
4. How each in-flow and out-flow will be referenced by the process' behavior, and
5. How a process will consume the data provided by an in-flow so that the same data will not be erroneously used again.

Data flow objects pertinent to a process are found by looking up the process name in the relation table. The position of the process name (From-Object or To-Object) is used to determine if the flow carries data into or out of the process. Additionally, the process must provide to the controlling function the name of a process, or a list of processes, that can be executed next. The relation table is searched to find which processes can possibly be executed next. The To-Object name is added to the set of next processes if the current process name is in the From-Object

position and the Association-Object is a flow-object whose *flow-link* attribute is *proc-proc*<sup>4</sup>. For example, consider the following portion of the relation table taken from the Library Problem (See Appendix E):

From-Object	Association-Object	To-Object
DETERMINE-STAFF-TRANS	TRANSACTION-3	ADD-BOOK
ADD-BOOK	UPDATED-BOOK-1	BOOKS
ADD-BOOK	ICO	ADDING-BOOK

Table 5. Excerpt from the Library Problem Relation Table

The in-flows to a process are determined by locating all the rows in the relation table with the process name as the To-Object. For these rows, the Association-Object represents an in-flow to the process. Similarly, the out-flows of a process are found by locating the entries where the process name is the From-Object and the Association-Object is *not* ICO. In these cases, the Association-Object represents an out-flow. Each process "is composed of" at least one behavior and behaviors are found by locating the entries where the process name is the From-Object and the Association-Object is ICO. Using this approach, we can determine from the sample relation table illustrated above that TRANSACTION-3 is an in-flow to process ADD-BOOK, UPDATED-BOOK-1 is an out-flow from ADD-BOOK, and ADDING-BOOK is the behavior associated with ADD-BOOK. The relation table also indicates that the ADD-BOOK process is a process that potentially can be executed after the DETERMINE-STAFF-TRANS process.

<sup>4</sup>There are five allowable values for a *flow-link*: *proc-proc*, *proc-store*, *proc-entity*, *store-proc*, *entity-proc*. These identify the classes of OML objects that the flow connects: *proc-proc* indicates data flows from a process to a process, *proc-store* indicates data flows from a process to a store, and so on for process to entity, store to process, and entity to process, respectively. Process to entity flows are not modeled in OML because entities, in this case, are external objects. Flows to these objects represent some form of output from the system and are modeled using *display* statements in the process' behavior.

The way in which we modeled flow objects assisted us in resolving the remaining issues. Once a flow object related to a process has been found, the flow's *flow-type* attribute is checked to determine

- if the OML object associated with the process (the object connected to the process by the flow) is internal or external to the system, and
- if the associated object is a process.

The in-flows to a process are verified prior to the execution of the process' behavior to ensure the behavior has all its required information. Flow objects are instances of the entity-class indicated by the flow's *flow-data* attribute, and share the same name as the OML flow-object. This modeling of flows benefited the translation of processes in the following ways:

1. It provided a way for processes to validate the contents of a flow object. Since flow objects are modeled as objects in the executable specification, the validity of the data flow can be determined by checking the flow's attributes. If the flow's attributes are undefined, then the flow is invalid. If a flow's attributes are defined, then the flow is valid and the process can consume the data.
2. It provided a naming convention to allow the specifier to address flow objects and their attributes while specifying behaviors.
3. After the data from an in-flow is consumed by the process' behavior, the process can invalidate the in-flow by un-defining the flow's attributes.

This translation methodology currently places the responsibility for using correct entity and attribute names in each process behavior with the specifier. This methodology was adopted to simplify the initial process-based translation and to allow us to focus on ways to compose the processes into an executable specification. As research in this area continues, more intelligence can



be added to the translation software or to the requirements elicitation tool to assist the specifier in writing the specification.

An alternate methodology for translating process objects would be to parameterize the function calls the process function makes to its behavior functions. When the process is translated, it would still check all the in-flow data to ensure they are valid, but these flows would also need to be passed as parameters to the Behavior function. Additionally, the out-flow data would also have to be received by the process from its behavior function. This methodology would result in increased complexity in the translation of process and behavior objects. To automatically parameterize the behavior function, the translation software would have to collect all the identifiers from the right-hand-side of assignment statements and all the identifiers from expressions in the behavior rules, and match their names to entity classes. These would form the in-flow parameters. A method would also be needed to differentiate an in-flow containing data from a query to a store, which is translated directly. The translation software would derive the out-flow parameters from the left-hand-side of assignment statements. This leaves two remaining problems. First, parameters are passed in order. To avoid run-time problems, the process would have to "know" the order of the in-flows expected by the behavior function. Next, the information about the behavior must be known at the time the process is translated. This could be established by translating the behavior first. However, because the REFINE executable specification is produced sequentially, the information concerning the parameter types and order would have to be added to the AST, so that the process translation function would have access to the information later. By translating flow objects to global variables, we eliminated the need for parameters and avoided the difficulties mentioned above.

**5.4.7 Flows.** The main purpose of data flows is to transfer data from one entity to another. OML uses flows to capture data items that are produced in one object and consumed in another. Flow-data associated with a flow has a data content, a data source type, and a data sink type which can be modeled in the flow-object's specification. How and when flows will be produced and

consumed is partially captured in the behavior specifications of the processes associated with the flow. It is also partially dependent on the order of execution of the processes represented by the DFM. Therefore, data flows in the OML specification have to be translated into a form that can be checked for valid information and which can maintain the information until it is consumed by another process. Flows can be modeled either implicitly or explicitly. Passing parameters between process functions and behavior functions is equivalent to modeling flows implicitly. Alternately, flows can be modeled explicitly as global objects (variables) that behavior functions can update as necessary. (26:122) Three basic issues influenced our decision on how to translate flows:

1. We needed a method that would allow the translation software to identify the specific flow object that a process' behavior was referencing.
2. We needed to ensure the data values that were produced and placed in flows would persist until the controlling function called the process function that would consume the data. (An associated concern was the development of a control architecture to execute process-oriented specifications that would not generate an excessive number of nested subprogram calls.)
3. We needed a mechanism to indicate whether the data in the flow was valid.

A flow in an informal model can be characterized by the data it carries, and by its source and destination endpoints (the process, store, or terminator that produced it and the process, store, or terminator that will consume it). In OML, details about exactly what data values should be placed in the flow are contained in the behavior specifications of the process with which the flow is associated. Exact data values will usually not be known until run-time. For this reason OML's flow objects have to provide a template for the data communicated between entities which can be filled in as the specification executes.

An OML flow object is composed of a *flow-data* attribute and a *flow-link* attribute. The *flow-data* attribute references an entity class. This attribute defines the type of data the flow will carry when passing data from its source to its sink. We needed to decide which representation of

flows would be more suitable in REFINE: global variables in the REFINE executable specification, or variables local to the appropriate process and behavior functions. Since OML processes "know" what flows are associated with them, the processes can collect all of the data required to allow its behavior to execute. Once collected, the process can verify the data is valid and pass the necessary data to its behavior function. Using this method, the flow data can be local to a process. On the other hand, because data can be passed from one process to several other processes (as well as between the process and its behavior), and because only one of the receiving processes can execute at one time, flows need to exist unaltered between one process and another. Therefore, although data flows can be modeled either by local or global REFINE objects, global objects provide three advantages:

1. Global objects provide a fixed group of names that can be used to reference the data produced and consumed in behavior specifications.
2. They provide an object that can contain the information produced by one process until the process that consumes it is executed, in essence maintaining the state of the system.
3. The validity of the data contained in global flow objects is easily determined by interrogating the object's attributes. If the attributes are undefined, the data is invalid. Conversely, the data is valid if the attributes are defined.

When an OML flow is translated into the REFINE specification, the flow is instantiated as an object of the *flow-data* attribute type using the REFINE *make-object* command. During translation, only the name attribute is given a value. The rest are left undefined. These attributes are set when the specification is executed by behavior objects that produce data and fill out-flows to be consumed by another process.

The *flow-link* attribute identifies the two types of objects that the flow connects. The specific objects that a flow links together are listed in the relation table. Flows can only connect certain combinations of objects. Legal links can be formed between the following object pairs: Process-

Process, Process-Store, Process-Entity, Store-Process, and Entity-Process. These categories of information are useful in the translation of processes and behaviors, but are not translated directly into the REFINe specification. This information is used as follows:

- Process-Process flows indicate that a global flow variable needs to be instantiated to pass information between processes.
- Process-Store flows also indicate a need for a global flow to be created. It is used to collect data produced by a process that will be added to a store.
- Process-Entity flows represent data that is displayed.
- Store-Process flows represent processes that retrieve information from a store. This type of flow does not change the entity or the store object. For this reason, OML's basic expression capability was enhanced to provide a group of function calls to perform operations on sets (since stores are converted into sets) and to display information.
- Entity-Process flows represent information that is input to a process from outside the system. These flows are represented by global objects and indicate to the control architecture that the user must be prompted to provide information to fill this flow.

*5.4.8 Stores.* OML stores are collections of entities designed to emulate the stores represented in Data Flow Models (DFM). Stores are defined by their nature, content, key, and order. The *nature* of a store defines its ordered-ness. Set-natured stores are unordered and have no repeated elements. Sequence-natured stores are ordered and may have repeating elements. Sequence-natured stores are ordered on their *key* attribute, and are arranged in ascending or descending order based on the store definition's *order* attribute. The *content* attribute of the store identifies the class of instances that will be placed in the store. The content of all stores must be homogeneous, consistent with accepted DFM rules (23:127).

The translation software currently does not implement all OML store features. All stores are translated as sets regardless of the nature attribute specified. This implementation was selected for two reasons.

1. Investigating how to develop and integrate the components of a process-oriented specification into an executable specification was more important than investigating alternative implementations for stores. Therefore, we constrained the complexity of the translation problem by focusing on sets. Because REFINE uses different instructions for set manipulation and sequence manipulation, new OML functions (in addition to *union* and *set-diff*) would need to be added to OML to provide sequence-oriented operations. Alternatively, the existing operations could be translated in two different ways (for ordered and unordered stores), depending on the context in which they are used. This would increase the complexity of the specification's behavior translation routines. As the specification's behavior is being translated, the translation software would have to determine the nature of the store being operated on and select the appropriate store operation to insert. Additionally, new OML functions would have to be provided, or automatically generated by the translation software, to insert items into ordered stores as specified by the key attribute and ordering attribute.
2. Including the details about stores discussed above is beyond the scope of a specification tool. The nature of a store and how it is accessed are implementation issues (29:154). The specifier should be concerned about the existence of stores and their association with processes, but not with the ordering of data in the store. Those factors should be considered later in the design phase. Modeling stores in such great detail *can* be done, but detracts from the level of abstraction, shifting focus from *what* should be done to *how* it should be done. It also increases the complexity of the translation software without adding any tangible benefit from a validation perspective.

Modeling the activities that deal with stores was also a challenge. Several factors required resolution:

1. Modeling data items as REFINE objects created problems with retrieving and modifying objects in the stores (to avoid loss of information when updating an existing object). In REFINE, the statement  $(A = B)$  is meaningless if A and B are objects. If the intention of the statement  $(A = B)$  is to copy all of B's attribute information into A to make the two objects identical, then it must be done explicitly. We devised a means for the translation software to detect when a behavior is attempting to assign one object to another and created a library function to copy one object's attributes into another. The object manipulation functions (see Appendix C, Section C.2) written by Capt Mary Anne Randour have helped immensely with the problem.
2. When generating the REFINE executable specification, we made all the objects in the data architecture (entities, flows, and stores) passive. Because stores have no behavior, they cannot add or delete information passed to them in a flow. Therefore, we needed to examine how data would be added to, deleted from, or updated in a store. Two options existed:
  - (a) In a DFM, a read from a store is modeled as an unlabeled arc from the store to the process (23:127). An addition to, deletion from, or update of an element in a store is modeled as an arc from the process to the store labeled with the type of data it carries. One method for modeling store-related activities is to model all the flows in OML as they appear in the DFM. The control architecture will detect flows to and from stores, and make the appropriate changes to the stores. This option requires the development of a library of access operations to act on a store. The translation software would need to be able to discern, from the structure of the behaviors, which operation (add, delete, or update) needed to be performed on the store. The translation software would then have to decide if the store-related action involved an in-flow or out-flow so that the control

process could place the information from the store into the flow (or vice versa) at the appropriate time. That is, if data is required to be retrieved from a store, then it is associated with an in-flow to the process being modeled and must be retrieved prior to the execution of the process' behavior.

- (b) A second option is to add expressions to OML that allow a specifier to perform operations on the stores directly. When behavior objects modify the contents of a store, flow objects normally record this transfer. Process specifications (p-specs), a part of the DFM, normally describe what action are performed on a store as well as describing the behavior of the process. By adding set manipulation functions to OML, OML behavior specifications can imitate the DFM p-spec by assigning data to its out-flows and then using its out-flows and the set operations to modify the store. This relieves the control architecture of the need to perform store operations. Using this scheme, flows exiting from stores can be eliminated from the OML specification because the access to a store is actually performed in the behavior. Flows to the store must still be modeled to provide the translation software a variable name and association between the process and the store.

We selected the second approach (option (b)) to keep the control architecture as simple as possible. To add an item to or delete an item from a store, the specifier uses *union* or *set-diff*. An item can be retrieved from a store by using the *getitem* command and then modified by changing the appropriate attribute values. This option also reduces the amount of information that needs to be extracted from the AST and tabularized for use during behavior translation. These functions are used when flows from a process to a store are being modeled. Flows must be specified in these cases to provide a connection between the process and the store. Testing for the existence of an item or condition in a store is accomplished using existential and universal quantification (*exists* and *forall*). In these cases, the flow doesn't need to be modeled because no information is actually moving out of or into the store.

*5.4.9 Relation Table.* Relation tables are a necessity when modeling a problem in OML.

Relation table objects are provided in OML to capture the associations between two objects in the system. The information contained in these tables is frequently used by numerous functions in the translation software, but relation tables are not directly converted into a REFINE construct during the translation. Relation tables are used to support the translation of other OML objects into REFINE and also to establish the sequencing of operations in the executable specification. Relation tables were used to find the following information:

- During the translation of each State object:
  - The set of all external events applicable to the state
  - The behaviors associated with each external event
  - The behaviors associated with the state
- When converting each Process object:
  - The set of in-flows required for the process to execute
  - The behaviors associated with the process
  - The set of processes<sup>5</sup> that can be executed after the current process has completed execution
- When translating behavior objects for state-based models, the next state to execute is determined from the relation table if an internal event name was provided as part of the behavior object.

By localizing all object association information in one table and removing the need for referential attributes in objects, we have developed a modularized, decoupled representation of the

---

<sup>5</sup>These processes are identified by determining what processes are connected to the out-flows. The information that will be entered into the out-flows during execution is entered directly by the process' behavior. Hence, the requirement for flow objects to have unique names. The out-flows themselves are not grouped and checked because they will be checked as the next process' in-flows.



objects in a system. From an object-oriented analysis and design point of view, this is very beneficial. However, from an implementation point of view, this representation can result in a significant degradation in the performance of the translation software. This representation requires the translation software to perform a significant amount of searching through the relation tables. As the size of a relation table grows, so grows the searching time required to find the desired information in the table. Therefore, a large relation table can result in a slower conversion process. A potential solution to this problem is to create multiple tables that only hold specific information. This can be accomplished by adding an attribute to relation table objects that indicates the type of associations contained in that table. For example, an *association-type* attribute could be created which can have the following values: flow, event, relationship, or ICO. A *flow* value for this attribute would indicate the table contains associations between objects that are linked together by flow objects. A similar meaning would apply for an *event* or *relationship* value. An ICO value would indicate the table contains associations between an object and its behavior. Such a modification to relation table objects should not be difficult to implement and should result in improved performance for the translation software.

**5.4.10 Constraints.** The Object Modeling Language allows the specifier to express limiting conditions on the objects in an OML specification. This is accomplished by instantiating the constraints section of an OML object. OML's design was heavily influenced by RML (15). Greenspan felt that a requirements language should allow the specifier to formally constrain the system he was describing. To do so, RML was developed with an assertion class to describe constraints. OML has no assertion class, but does provide expressions that can be used to describe constraints. What additional constraints should be described in the constraints portion of each object? Constraints currently can describe situations that must exist in terms of entities and their values. OML may need to be expanded to allow the specifier to describe timing dependencies and

functional dependencies. Constraints can be applied to each OML object type in the following manner:

- **Entities.** Additional constraints on entities may be a duplication of the information already expressed in the entity's range. However, constraints could possibly be used to express interrelationships that must hold between two or more entities.
- **Processes, States, and Behaviors.** For these objects, constraints are a means of expressing conditions that should remain invariant throughout the execution of the function representing the object. When and how these constraints should be used requires further investigation.
- **Stores.** Constraints on stores could be used to limit the size or data content of a store, although the usefulness of such constraints requires further research.
- **Relationships.** Defining constraints on relationships appears to be meaningless. However, the actual relationship may be useful as a type of constraint. (See Section 5.4.2 for a complete discussion.)
- **Events and Relation Tables.** No meaningful constraints can be proposed for events or relation tables that are not already provided by other objects in the system.

We recommend that constraint checking be performed on all passive objects (entities, flows, and stores) both at translation time and during execution. Additionally, constraint checking on dynamic objects (states, processes, and behaviors) should be performed prior to, and immediately after execution of the object function during the executable simulation. These constraints can be implemented as if-then test conditions in each function.

Some knowledge-based code generation and synthesis tools construct executable programs from libraries of predefined components. These systems use constraints as a means of restricting the number of library components that can be used in composing the solution to a specified problem. Because OML translates the specification directly, and does not assemble library components, this

use of constraints has a limited application. To do this at some future time would require the definition of standardized interfaces and definitions of all REFINES specification objects so that a library of components could be assembled. Then constraints could form heuristics that would limit the search and matching functions that would select suitable behaviors from the library. Adding this feature would expand the complexity of the behaviors that OML could model and reduce the amount of work required in specifying behaviors afresh with each specification. Ultimately, domain libraries could be developed containing the most basic building components that could be assembled with new processes and states to define new systems.

### 5.5 *The Value of Executing a Specification*

Executing a specification can help the specifier answer several important questions, such as:

- Does the specification contain any contradictions?
- Does the specification allow undesirable side effects or unspecified (but desirable) circumstances to exist?
- Are user interaction functions or any other "helper" routines needed?
- Are there any ambiguous requirements in the specification?

Executing the Home Heater and Library specifications helped us to locate several flaws in our initial definitions of the two problems. The following are some of the problems that were uncovered by executing the Home Heater specification:

- We discovered that the IDLE state was specified inconsistently. Our initial state-space definition for the IDLE state required the air temperature to be greater than the temperature which causes the heater motor to start (i.e.,  $AIR.temp > CONTROLLER.tr - 2$ ). This seemed reasonable since the system should be idle if the air is warm enough. To enter the MOTOR-ON state, the AIR.temp must be less than  $CONTROLLER.tr - 2$ . While in the MOTOR-ON, the

MASTER-SWITCH-OFF event occurred, turning the MASTER-SWITCH off. This caused the system to transition into the OFF state. When the system transitioned from the OFF state back to the IDLE state, the AIR.temp had not changed. It was still less than CONTROLLER.tr - 2 and a state-space violation occurred.

- A similar problem occurred with the HOLD state's state-space. Originally, we specified that the water-valve should be opened in this state. Testing the specification revealed that this was an incorrect requirement.
- Developing the user interface to the executable specification showed that external events needed to be specified to fully describe the connection of the system to its environment.
- Execution showed that we were missing the ABNORMAL SHUTDOWN state. HOLD's state-space required both the fuel-flow and combustion sensors to be unsafe. However, the system could enter into the HOLD state if the MASTER-SWITCH was turned off during the RUNNING state. Thus, the system was in the HOLD state but the sensors were safe which caused an error to be raised.

These are just a sample of the problems which were discovered by testing the executable REFINESPECIFICATION that was automatically generated from the OML Home Heater specification. Clearly, executing the Home Heater specification enabled us to produce a more consistent, unambiguous, and functionally correct specification.

Executing the Library specification also revealed many of the same types of problems mentioned above. However, it was also very useful in validating (in a limited sense) our translation software. One particularly important discovery was the necessity for all post-conditions in a behavior rule to be mutually independent. This requirement is very important since we originally designed the translation software to convert each pre-post-condition rule into a REFINESPECIFICATION transform construct. The order in which the post-conditions of a REFINESPECIFICATION transform are satisfied cannot be determined by the specifier, therefore each post-condition statement must be independent. This issue surfaced

when the executable specification terminated abnormally while executing the CHECKING-BOOK-OUT function, because it tried to modify an attribute of an object that was not yet retrieved from a store. In this case, executing the Library specification not only revealed problems with the Library requirements, but also exposed important performance restrictions regarding the translation software.

### 5.6 Summary

This chapter has described the rationale used in constructing the software that automatically translates an OML specification into an executable REFINE specification. The translation is accomplished by a multi-step process:

1. The OML specification is parsed into a REFINE Abstract Syntax Tree (AST). This capability was provided by developing OML's domain model and grammar, which interface with DIALECT, REFINE's compiler generation tool.
2. The OML objects contained in the AST are converted into equivalent<sup>6</sup> REFINE constructs by the translation software.
3. A controlling function is added to the converted file by the translation software to produce an executable specification.

Converting a process-oriented informal specification into an executable specification proved to be more difficult than converting a state-based informal model. This was primarily true because the process-oriented specification did not include any control information. In spite of the difficulties encountered, the translation of both the state-based and process-based models into executable specifications was very successful. The executable specifications derived from the Home Heater

---

<sup>6</sup>While there is no one-to-one mapping for all OML objects to REFINE objects, we feel that we have captured the essential meaning of each particular OML object in REFINE and that, in this conceptual sense, the constructs are equivalent. Translation of larger, more complex problem into OML should verify the correctness of our conceptual mappings from OML to REFINE.

and Library problems exposed several inconsistent and incorrect requirements in their respective informal specifications. Further, the translation process is very simple to perform and easily accommodates re-translation of modified specifications. Upon discovering an error, an OML specification can be corrected, and then converted to an executable specification with one command. From a specifier's point of view, it was easy to focus on the specification alone and remain detached from implementation issues. The Object Modeling Language and the translation software have shown that the gap between informal and formal specifications can be spanned. The two techniques complement each other: one aides the specifier in conceptualizing information, the other provides the formality needed to remove ambiguity. Automation can assist the specifier in developing formal specifications and maintaining the rigor necessary to build systems that meet these specifications.

## *VI. Conclusions and Recommendations*

### *6.1 Objectives and Results*

The goal of this thesis was to develop a method for transforming the information contained in informal software specifications into an executable formal specification that can be used to verify expected system behavior and serve as a basis for formal software derivation.

This research has successfully accomplished this goal by developing a method for bridging the gap between informal and formal specifications. By developing a method for translating an informal specification into an executable formal specification, users, specifiers, and, developers can take advantage of the benefits offered by formal specifications. One direct benefit of this research is that it provides a method for revealing requirement errors at a very early stage in the software development lifecycle. The following objectives were established to help us achieve our goal:

1. To establish a minimal set of constructs that represent the content and behavior of informal analysis models, specifically Entity Relationship Models (ERM), Data Flow Models (DFM), and State Transition Models (STM).
2. To develop a methodology for translating the information contained in these informal models into a formal object-based language.
3. To develop a tool to translate formal, object-based specifications into an executable environment.

This research produced the following results which directly contributed to meeting our objectives:

1. The development of the Unified Abstract Model (UAM) to provide a unified representation of entity relationship, state transition, and data flow models.
2. The development of the Object Modeling Language (OML) which directly models the objects in the UAM and which has a formal language notation amenable to automatic translation.

3. The development of a translation tool to convert an OML specification into an executable specification.

We accomplished our first objective by defining the Unified Abstract Model (UAM) described in Chapter III. The UAM was developed to provide a unified, object-oriented representation of all the components (objects and attributes) necessary for modeling the information contained in DFMs, STMs, and ERMs.

The Object Modeling Language (OML) defined in Chapter IV is the bridge that spans the gap between informal and formal specifications. Our review of currently available specification languages (see Chapter II) was intended to locate a specification language to support our Unified Abstract Model. However, we did not find a specification language to directly and naturally support the UAM. For this reason, we developed OML. OML directly supports the objects and attributes defined in the UAM and provides a formal language notation that is easily parsed into an Abstract Syntax Tree (AST) representation. Therefore, OML provides a structured notation for naturally specifying informal specifications which is easily translated into a formal object-based representation.

The translation software developed during this research successfully parses an OML specification into an AST and then converts the information contained in the AST into an executable REFINE specification. Through the execution of a specification, the user can now validate at a very early stage of development, that his informal requirements specification correctly and unambiguously captures his intentions. The translation software is described in Chapter V and is provided in Appendix C.

Additionally, the REFINE Software Development Environment proved to be a very beneficial development tool. DIALECT, REFINE's language manipulation tool, enabled us to quickly develop a compiler for translating an OML specification into a REFINE AST. We believe that DIALECT was easier to learn and understand than other popular compiler tools, such as Lex and YACC, because



DIALECT's syntax enabled the compiler software to very closely resemble the Backus Naur Format description of the parsed language without requiring cryptic notations. Furthermore, REFINE's high level constructs significantly simplified the amount of effort needed to convert the information contained in the AST into an executable specification. For example, REFINE's universal and existential quantification capabilities allowed us to perform operations over a group of objects without requiring us to produce detailed code for searching the object-base.

## *6.2 Recommendations for Future Research*

This thesis was very successful in defining and implementing the techniques required to translate an informal software specification into a formal executable specification. This section offers several recommendations for future research. They are categorized into three groups: improvements to the existing translation tool, additions to the translation tool, and supporting research.

### *6.2.1 Improvements to the Existing Translation Tool.*

- *Augment the hierarchical structuring capabilities of OML.* OML should be expanded to allow it to model more complicated object hierarchies. For example, OML should be expanded to allow the modeling of classes of classes for entity objects and to allow the definition of classes for all other OML object types (e.g. classes of states, classes of processes, etc.). Additionally, nesting of OML objects should be implemented to assist in modeling more complex problems.
- *Take advantage of multiple relation tables to improve translation performance.* Currently, OML allows the specifier to define one or more relation tables. However, there is no mechanism for knowing what kind of relationships are contained in each relation table. Multiple relation tables that contain specific types of relationships can significantly improve the performance of the translation software by reducing search time through the relation tables. This suggestion is further discussed in Section 5.4.9. Other enhancements should be investigated to improve the performance of the translation software.

- *Augment the translation software to sequence events based on priority.* OML allows the user to specify a priority attribute (optional) for each event object. The priority should be used to determine the sequence in which events occur if more than one event is eligible for processing at the same time. The priority attributes should be used during the translation to guide the execution of events.

#### 6.2.2 Additions to the Translation Tool.

- *Define a controlling function for OML specifications that contain both state-based and process-based information.* Future research should investigate how state-based and process-based information can be integrated together to develop one executable specification. This capability would be beneficial to Real-Time Structured Analysis (RTSA) which is a widely used technique for specifying embedded computer systems. Currently, a different controlling function is added to the executable specification for state-based models and process-based models. A third controlling function should be developed for cases when both state-based and process-based information exist in the same OML specification. Additionally, while developing the process translation routines, we noticed numerous instances where states and processes are similar. It may be possible to combine state and process objects into one object, but a more thorough analysis needs to be accomplished to evaluate all the side effects of such a fundamental change to the UAM and OML. If this combination is possible, then the two controlling functions would have to be modified or merged together.
- *Add semantic checking of OML specifications.* Future research should determine if the semantic checking should be performed by the translation software or by the elicitation tool. Section 4.3 discussed the semantic requirements of an OML specification. Currently, OML specifications are not checked for these semantic requirements. For example, entity objects should be checked to ensure that they are assigned legal values (i.e. correct type and within specified range).

- *Determine if OML should allow store objects to be specified as sequences.* Currently, OML allows the specifier to define the ordered-ness of a store object. Future research should investigate whether a specification should stipulate whether or not the elements in a store are ordered. This issue is discussed further in Section 5.4.8.
- *Fully incorporate PDL as an option for specifying behaviors.* PDL is very important to OML for allowing the realistic specification of continuous type behaviors. We have two suggestions for accomplishing this task. Currently, the PDL provided in Appendix B is a subset of the Ada language. We have integrated PDL's domain model and grammar in with OML. However, further modification must be made the PDL semantic checking and translation software as discussed in Section 5.4.5. Once PDL is fully implemented in OML, the translation of pre-post-condition behaviors should revert back to a REFINE transform construct. As a second option, we suggest that a subset of REFINE be used as the PDL standard instead of Ada PDL. In this case, the PDL would already be executable in the REFINE environment. An evaluation would need to be performed to determine the minimal, most useful constructs required to fully specify behavior. The advantage of this approach is that REFINE is a wide-spectrum language and can specify sequential behaviors (that pre-post-conditions cannot) in more general terms than Ada.
- *Incorporate the translation of OML object constraints.* OML allows constraints to be specified for each object. Future research should determine what the constraints section of each OML object should be used for, and how the constraints should be translated into the executable specification. Section 5.4.10 provides several suggestions concerning the application of constraints for each OML object and how they could be handled by the translation software.

### 6.2.3 Supporting Research.

- *Implement the direct simulation method for executing an OML specification.* This thesis identified two methods for executing an OML specification. The method we pursued was the

translation of an OML specification into a REFINE specification (source code) which can be compiled and executed in the REFINE environment. A major advantage of this method is the REFINE specification can serve as a basis for formal software design. The second method is to simulate the behavior of the OML specification by directly executing the information contained in the AST. We recommend this approach be pursued in future research. The second approach is more desirable for performance reasons. As the size of the OML specification gets larger, it takes a longer amount of time to generate REFINE source code and compile and load it than it would take to simply execute the AST. Further, this translation time also affects the amount of time required to modify an OML specification and re-translate it into an executable form. The direct simulation approach, however, does not produce a formal specification, and therefore does not support continued development. Thus, both methods have unique advantages and should be used to complement one another.

- *Implement more complex problems in OML.* The Library and Home Heater problems implemented in this thesis were limited in complexity. More complex test problems should be implemented in OML to further test its ability to capture the information contained in informal models. Additionally, relationship objects (modeled in ERMs) were not used during the translation of the Library and Home Heater problems into executable specifications. More complicated test cases may help in revealing the role that relationship objects play in the development of an executable specification.
- *Develop an elicitation tool to assist the specifier in constructing an OML specification.* OML enabled the translation of informal specifications into formal specifications to be automatable, but further support is needed to make construction of an OML specification a realistic task. This is an example of the need to formalize and automate the specification process as well as the specification. (11:52) The practicality of OML for modeling larger problems will likely be limited without the support of a front end tool. Further, an elicitation tool would consistently

develop a syntactically and semantically correct OML specification. Also, specifying a large problem manually in OML would be very tedious and prone to error.

- *Incorporate knowledge-based techniques.* Currently during execution, the specification provides very rudimentary information as to the source of the specification error. The executable specification “knows” if a state-space is incorrect or if a process has no in-flow data to operate on. However, it cannot tell the specifier how the error condition occurred. The specifier must rely on the OML specification, the informal model documentation, and his own ability to construct the path of change that led to the error. Valuable knowledge could be gained by researching the development of rules or procedures that will aide the specifier in locating the source of inconsistency in the specification. This idea could also be extended to include the development of a static correctness check that would occur prior to translation and execution of the specification.
- *Verifying correctness.* We have concluded that the translation process we developed is correct on the basis of testing. After developing the translation process, we tested and improved it while working through the Home-Heater problem. The basic translation methodology also worked for the Library problem. However, we had to expand the translation process to handle specific process-based information. We believe these two problems are representative of the type of information that the translation software will encounter. However, this does not verify that the translation process will work well for any case. Research needs to be done to develop a technique, possibly from graph theory, for verifying that the translation software does, in fact, map enough essential ideas from OML to REFINE for the REFINE specification to be considered a valid representation.

### *6.3 Concluding Remarks.*

Numerous past and current software development programs proclaim the need for software engineers to do a significantly better job of requirements analysis and specification. The increasing complexity of problems that software is being required to solve can only exacerbate the problems of current trends in software development. This thesis has identified a means for attacking one of software development's most difficult problems: correctly specifying software requirements. We have developed a process for converting a user's informal specification into an executable specification. By observing the behavior of an executable specification, developers can validate the accuracy of informal specifications and discover requirements errors prior to software development. This early discovery of errors will result in substantially lower software costs, decreased risk in development, and significantly improved software systems.

Furthermore, the formal, executable specification can serve as a basis for formal software design and thus further aid in the development of successful software systems. We have shown that informal and formal methods are complementary in nature, and we have provided a means for "bridging the gap" between them. The software engineering community must make the transition to formal methods in order to meet the demands of software development in the 1990's and beyond.

## Appendix A. Summary of OML Syntax and Semantics

### A.1 Syntax

A concise summary of OML syntax is given below using a slightly modified BNF notation.

Meta-symbols(unless quoted):  $\langle \rangle ::= \{ \} [ ] |$

Reserved words are in **bold face**

Terminal symbols are un-delimited words

Symbols consisting of a string in angle brackets (e.g.  $\langle \rangle$ ) are nonterminals

Required punctuation is denoted by “double quotes”

$[\langle \text{symbol} \rangle]$  signifies zero or one occurrences of  $\langle \text{symbol} \rangle$  (e.g.  $\langle \text{symbol} \rangle$  is “optional”)

$\{\langle \text{symbol} \rangle\}$  signifies zero or more occurrences of  $\langle \text{symbol} \rangle$

$\{\langle \text{symbol} \rangle\}^+$  signifies one or more occurrences of  $\langle \text{symbol} \rangle$

#### 1. Informal model

```
 $\langle \text{informal-model} \rangle ::=$   
    specification  $\langle \text{name} \rangle$   
     $\{\langle \text{analysis-object} \rangle\}^+$ 
```

#### 2. Analysis Object

```
 $\langle \text{analysis-object} \rangle ::= \langle \text{class-definition} \rangle \mid \langle \text{instance-definition} \rangle$   
  
 $\langle \text{class-definition} \rangle ::=$   
     $\langle \text{class-name} \rangle$  class-of entity  
    [parts  $\langle \text{user-declared-attr} \rangle$  {“,”  $\langle \text{user-declared-attr} \rangle$ }]  
    [constraints  
    { $\langle \text{expression} \rangle$ } ]
```

**<instance-definition> ::=**

**<object-name> instance-of**

**<instance-value>**

**[constraints**

**{<expression>} ]**

### 3. Instance Values

**<instance-value> ::=**

**<entity-fact> | <process-fact> | <state-fact>**

**| <store-fact> | <relationship-fact> | <flow-fact>**

**| <event-fact> | <relation-table> | <behavior-fact>**

**<entity-fact> ::=**

**( entity type : (internal | external)**

**[parts <user-declared-attr> {“,” <user-declared-attr>} ]]**

**| ( <class-name> [values <user-defined-attr> {“,” <user-defined-attr>}] )**

**<process-fact> ::= process**

**<state-fact> ::= state**

**state-space “.” <expression> {“,” <expression>}**

**<store-fact> ::= store**

**nature “.” set | sequence**

**content “.” <class-name>**

**[key “.” <attribute-name>]**

**[order “.” ascending | descending]**

**<relationship-fact> ::= relationship**

**type “.” ico | isa | general**

**cardinality “.” 1-1 | m-1 | 1-m | m-m**



**<flow-fact> ::= flow**

**flow-link “.”** proc-proc | proc-store | proc-entity  
| store-proc | entity-proc

**flow-data “.”** <class-name> | <object-name>

**<event-fact> ::= event**

**event-type “.”** internal | external

**[priority “.”** integer-literal]

**<relation-table> ::= relation-table**

<object-name> “,” <association-name> “,” <object-name>

{ “,” <object-name> “,” <association-name> “,” <object-name> }

**<association-name> ::= <object-name> | ICO | ISA**

**<behavior-fact> ::= behavior**

<process-description-lang> | <pre-post-condition> | <decision-table>

**<process-description-lang> ::= <ada-program><sup>1</sup>**

**<pre-post-condition> ::=**

<pre-condition> “- ->” <post-condition> **event** <next-event>

{ “,” <pre-condition> “- ->” <post-condition> **event** <next-event> }

**<pre-condition> ::= <expression> { “&” <expression> }**

**<post-condition> ::=**

<assign-stmt> | <function-call>

{ “&” <assign-stmt> | <function-call> }

**<decision-table> ::=**

< condition-row > { “,” < condition-row > } “- ->”

---

<sup>1</sup>See Ada PDL syntax Appendix B

```

[<action-row> {"," <action-row>}]
event {"," <next-event>}+

<condition-row> ::=
    <condition-variable> "," <condition-entry> {"," <condition-entry>}

<action-row> ::=
    <action-variable> "," <action-value> {"," <action-value>}

<condition-entry> ::=
    dont-care | <condition-value> | <condition-range>

<condition-variable> ::= <object-name> "." <attribute-name>

<action-variable> ::= <object-name> "." <attribute-name>

<condition-value> ::= <value>

<action-value> ::= <value>

<next-event> ::= none | <object-name>

<condition-range> ::=
    <predicate-oper> <argument> {<arithmetic-oper> <argument>}

```

#### 4. Expressions

```

<expression> ::=
    forall "(" <name> {"," <name>} ")"
    "(" <expression> {"&" <expression>} ">" <expression> ")"
    | exists "(" <name> {"," <name>} ")" "(" <expression> {"&" <expression>} ")"
    | "(" <expression> ")"
    | not <expression>
    | <expression> <connective> <expression>

```

| < condition>  
 | **true**

<connective> ::= **and** | **or**

<condition> ::= <term> <predicate-oper> <term>

<term> ::= <argument> {<arithmetic-oper> <argument>}

<predicate-oper> ::= < | > | >= | <= | = | ≠ | **in**

<arithmetic-oper> ::= + | - | *div* | \* | **set-diff** | **union**

<argument> ::= <condition-variable> | <value> | <setbuilder>

<setbuilder> ::= “{”<name> “|” <expression> {“&”<expression>} “}”

<getset> ::= “getset” <setbuilder>

<getitem> ::= “getitem” “(” <setbuilder> “)”

## 5. Statements

< assign-stmt > ::= <action-variable> “:=” ( <term> | <getset> | <getitem> )

< function-call > ::=

**create** “(” < object-name> “.” <class-name> “)”  
 | **destroy** “(” <object-name> “)”  
 | **display** “(” <object-name> | <setbuilder> “)”

## 6. User-defined attributes and names

User defined attributes are identifiers introduced by the user:

<user-declared-attr> ::=

<attribute-name> “.” <unranged-attr> |  
 (<ranged-attr> **range** “{”<enumerated-range> | <real-range> | <integer-range> “}”)  
 [**init-val** <value> ]

**<user-defined-attr> ::= <attribute-name> “.” <value>**

**<unranged-attr> ::= **boolean** | **string****

**<ranged-attr> ::= **integer** | **real** | **symbol** | **set** | **sequence****

**<enumerated-range> ::= [**<value>** {“,” **<value>**}+]**

**<real-range> ::= **real-literal** “.” **real-literal****

**<integer-range> ::= **integer-literal** “.” **integer-literal****

## **7. Terminal symbol productions**

**<value> ::=**

**integer-literal | boolean-literal | real-literal**

**| string-literal | symbol-literal | set-literal | sequence-literal**

**<name> ::= **string-literal****

**<attribute-name> ::= **string-literal****

**<class-name> ::= **string-literal****

**<object-name> ::= **string-literal****

## A.2 *Semantics*

This section defines the semantic requirements of OML.

### 1. General semantics.

- All analysis object names must be unique.

### 2. Entities.

- An entity object's range field defines the attribute's legal range of values.
- The initial values assigned to all entity attributes must satisfy the state space of the initial state.

### 3. Processes.

- None.

### 4. States.

- The first state in the OML specification is assumed to be the start state.
- The arguments used in defining the state space attribute must refer to existing entity attributes.
- The state space attribute must define the value or range of all object attributes that are important to the state.

### 5. Stores.

- The key and order attributes only apply to sequence-natured stores.

### 6. Flows.

- The flow-data attribute requires the class-name of the data that will be carried by the flow.

## 7. Events.

- External events must be associated with a behavior object to cause a change in the values of the state space objects.
- External events represent the actions a user can take during a simulation.

## 8. Relation-Tables.

- The ICO association is reserved for associations between a process, state, or external event and its corresponding behavior. While an ICO association object does not need to be explicitly specified, the object-ICO-behavior entry must be made in the relation table.

## 9. Behaviors.

- Behaviors must be explicitly defined for all state, process, and external event objects.
- The event field in behavior objects is only used by state behaviors and control process behaviors. External event behaviors cannot specify next events.
- If multiple behaviors are specified for one state, the state's behaviors must be listed in order of execution in the relation table. Each behavior will be executed in this order, and only the last state behavior should specify a next event.
- The variables used in the expressions and statements of behavior descriptions must be attributes of entity and flow objects. The entity attributes must be fully referenced by giving both the object name and the attribute name (e.g., object-name.attribute-name).

## 10. Expressions.

In the following discussion, S and X are sets and x is an element of a set:

- The *set-diff* operation requires two arguments, both of which are sets (e.g. S set-diff {x}). This operation removes the second argument from the first argument.

- The *union* operation requires the first argument to be a set and the second argument to be an element (e.g. S union x). This operation adds the second argument to the first argument.
- The *in* operation requires the second argument to be a set. This operation checks to see if the first argument is in the second argument.
- The *getitem* command locates a specific item in a store and allows the item to be modified, but does not remove the item from the store.
- The *getset* command locates a set of items in a store but does not remove the set from the store.

### A.3 OML Domain Model

The following figures show the “is a” hierarchy that exists among the domain objects. Rectangles are object classes. The lines labeled in lower case are map names from one object to other objects. Maps are unidirectional, therefore the object or group of objects that an object maps to are represented as rounded rectangles.

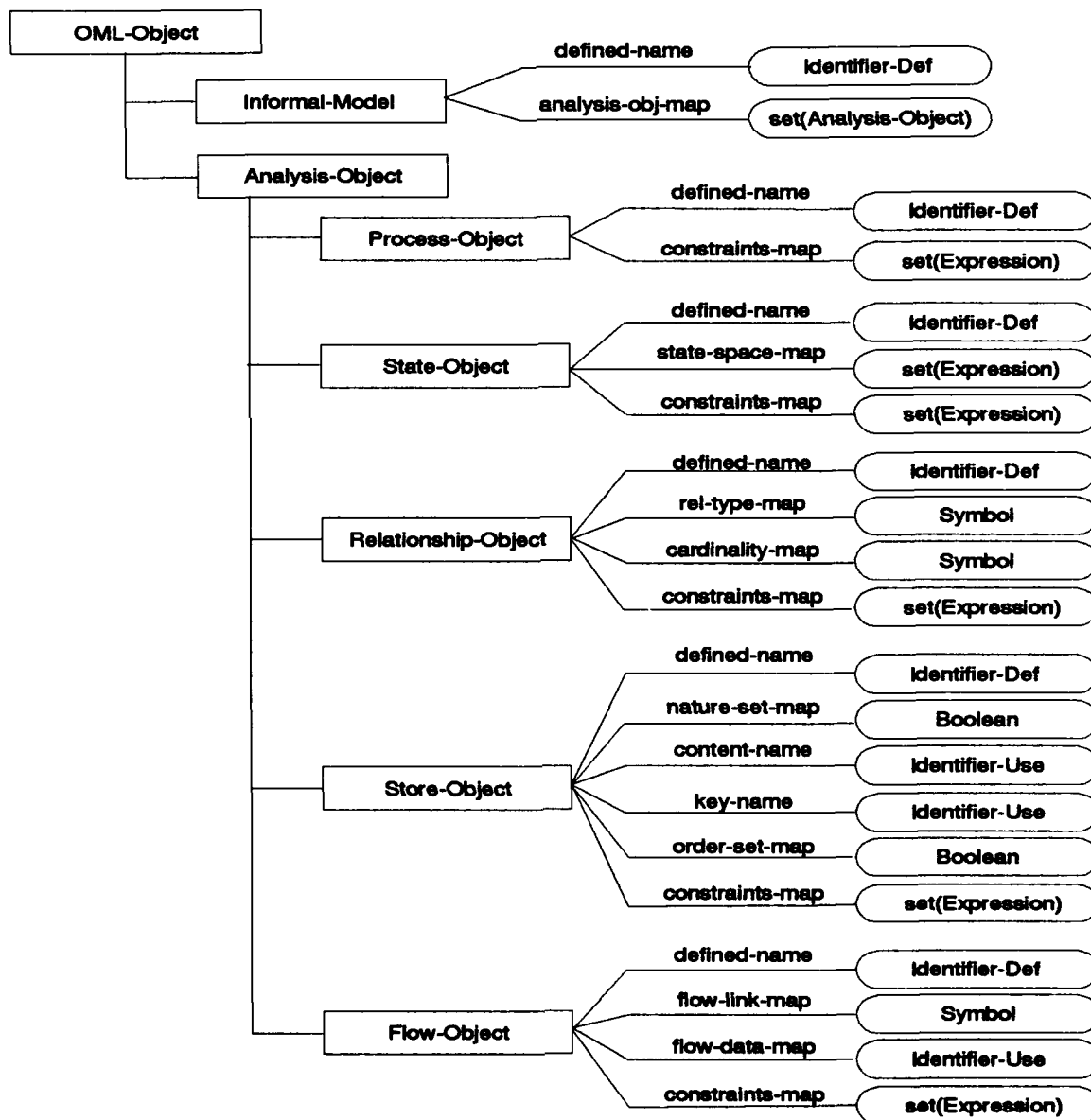


Figure 16. Hierarchy Detail with Object Mappings



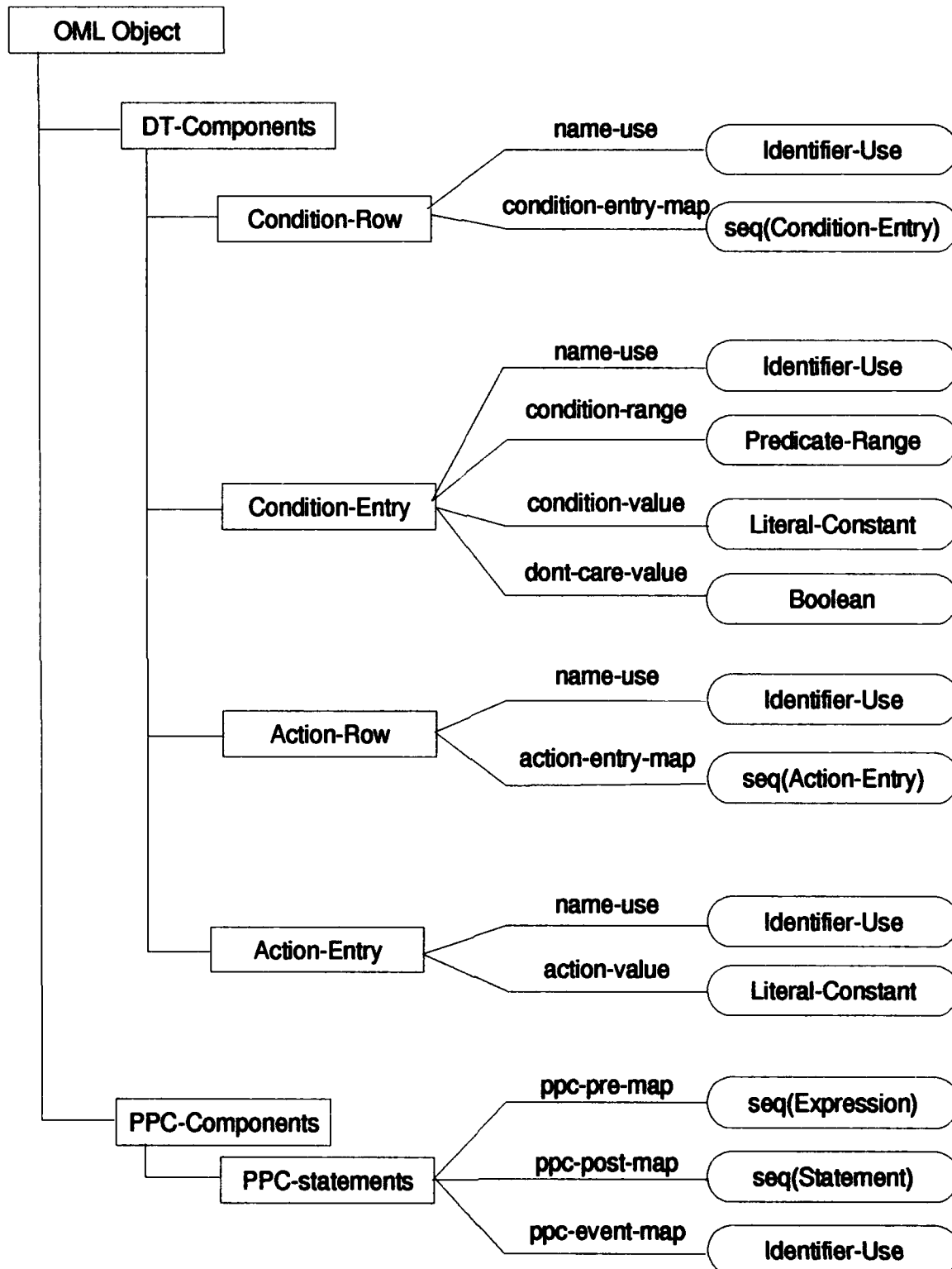


Figure 17. Hierarchy Detail with Object Mappings, Continued

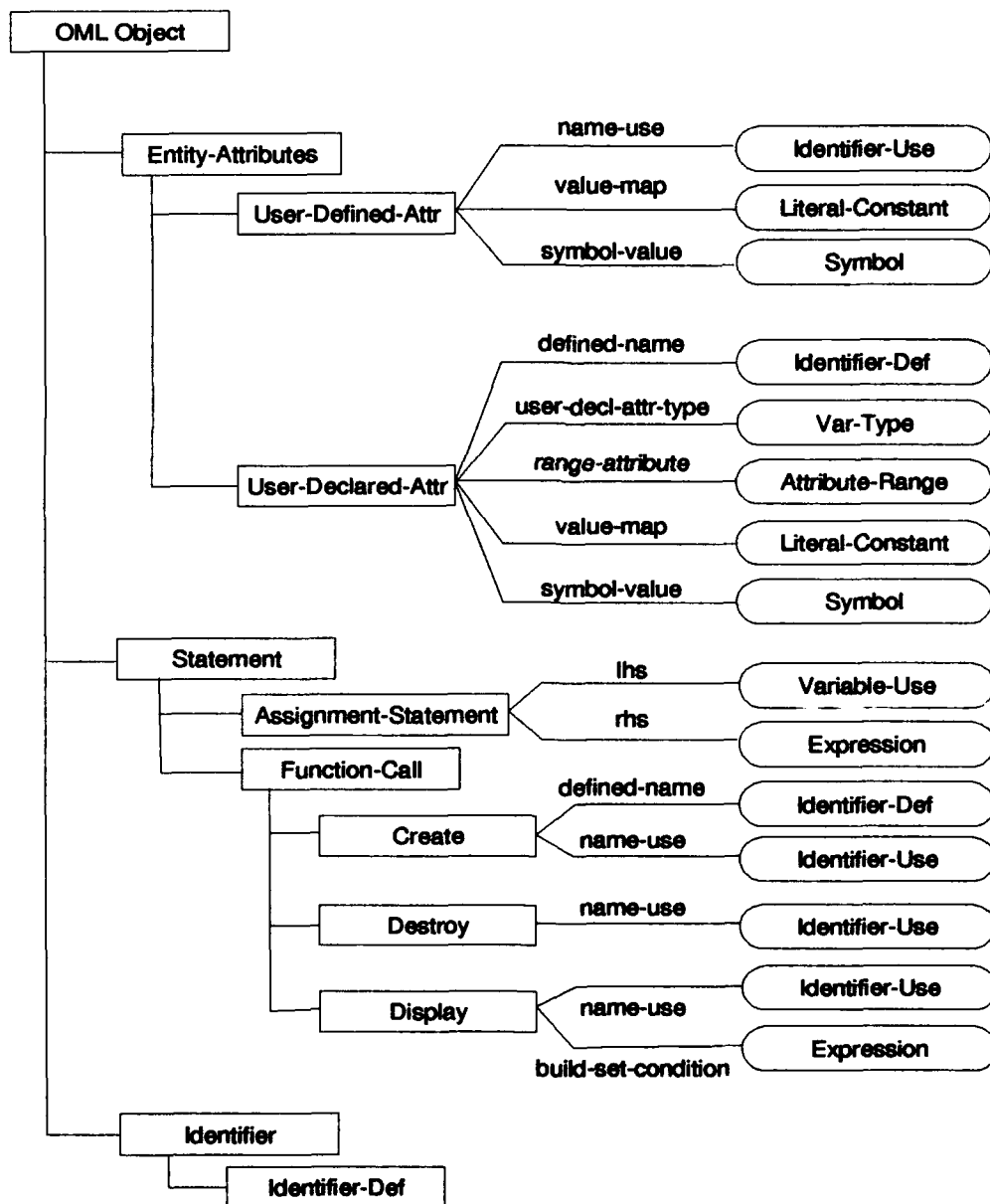


Figure 18. Hierarchy Detail with Object Mappings, Continued

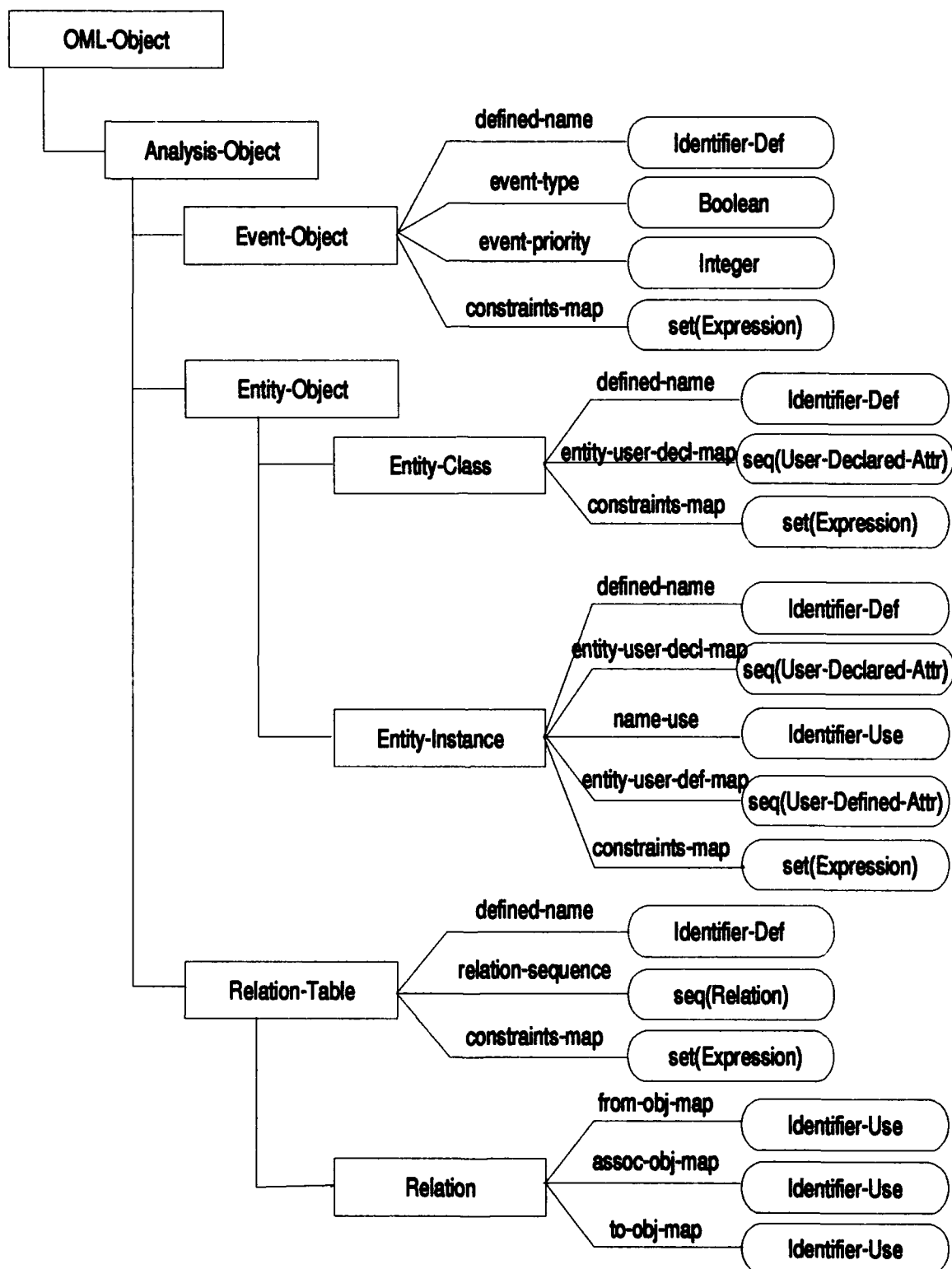


Figure 19. Hierarchy Detail with Object Mappings, Continued

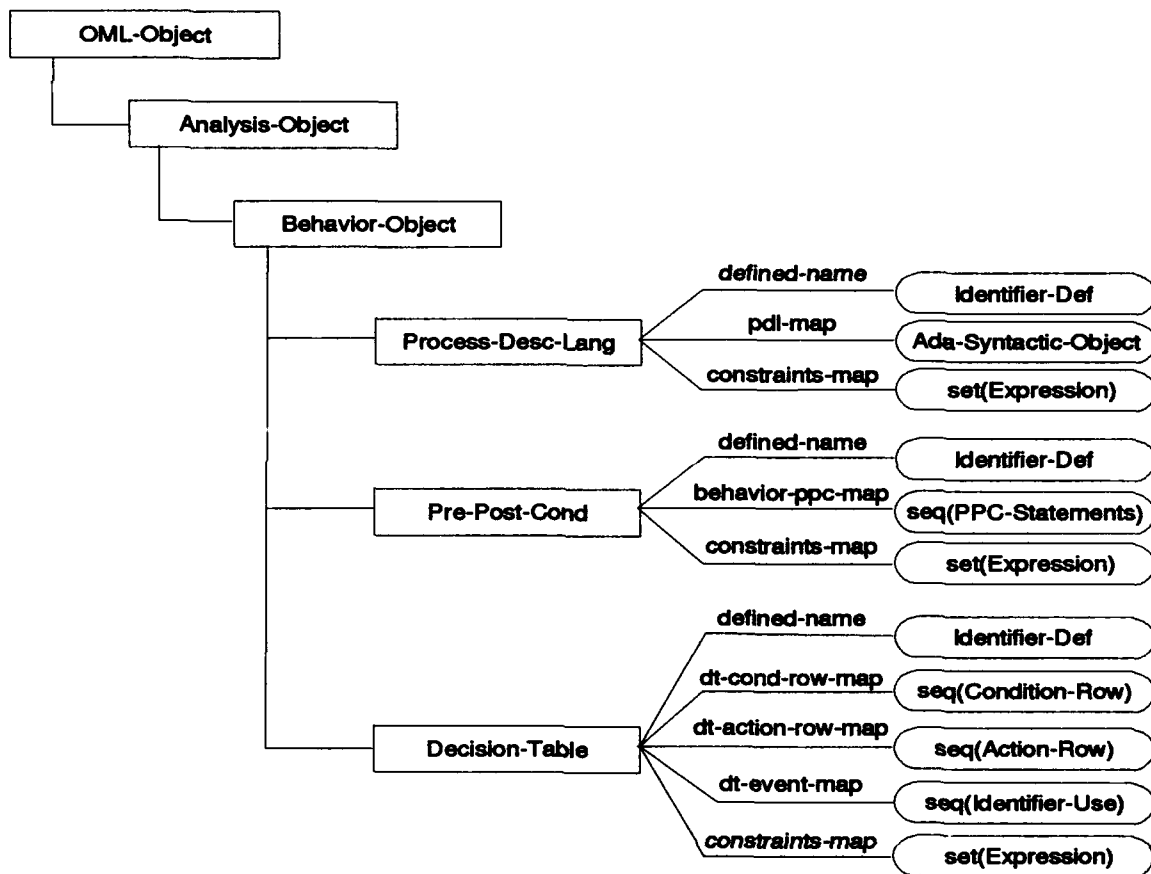


Figure 20. Hierarchy Detail with Object Mappings, Continued

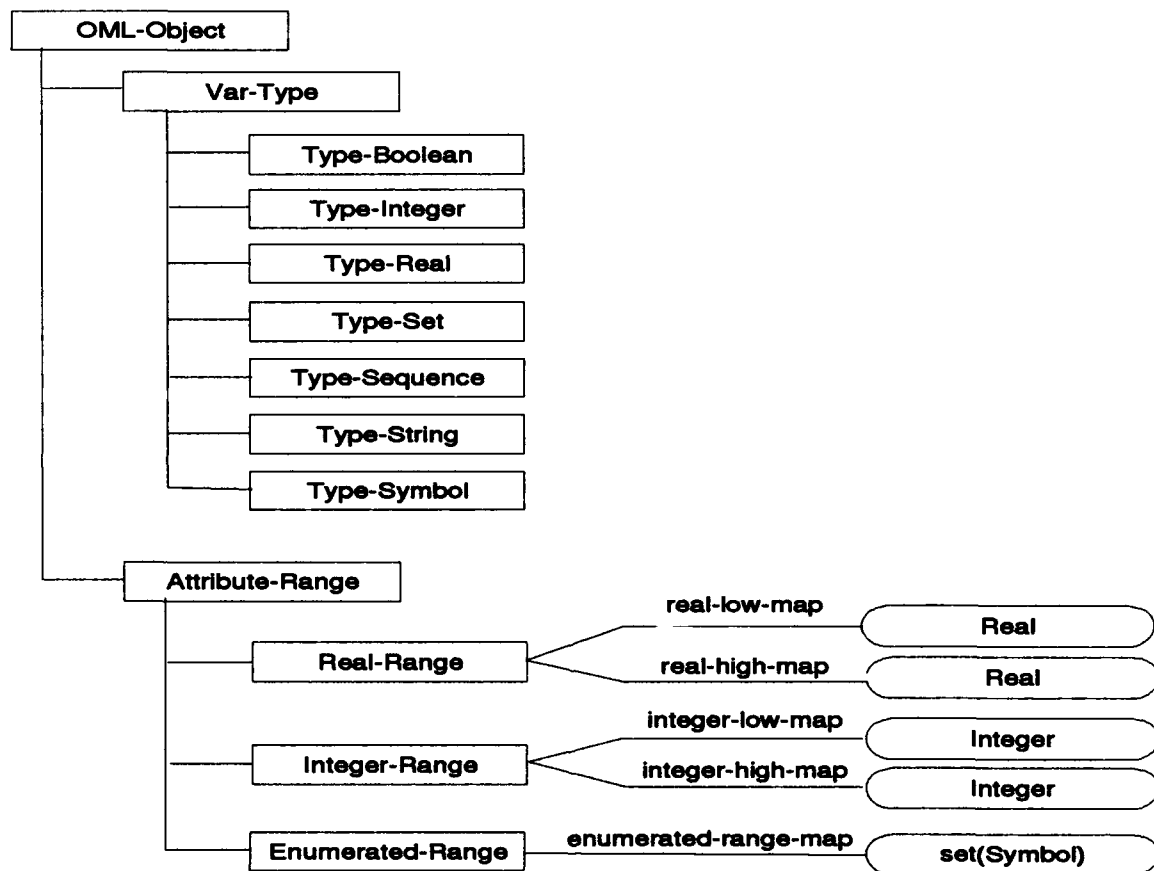


Figure 21. Hierarchy Detail with Object Mappings, Continued

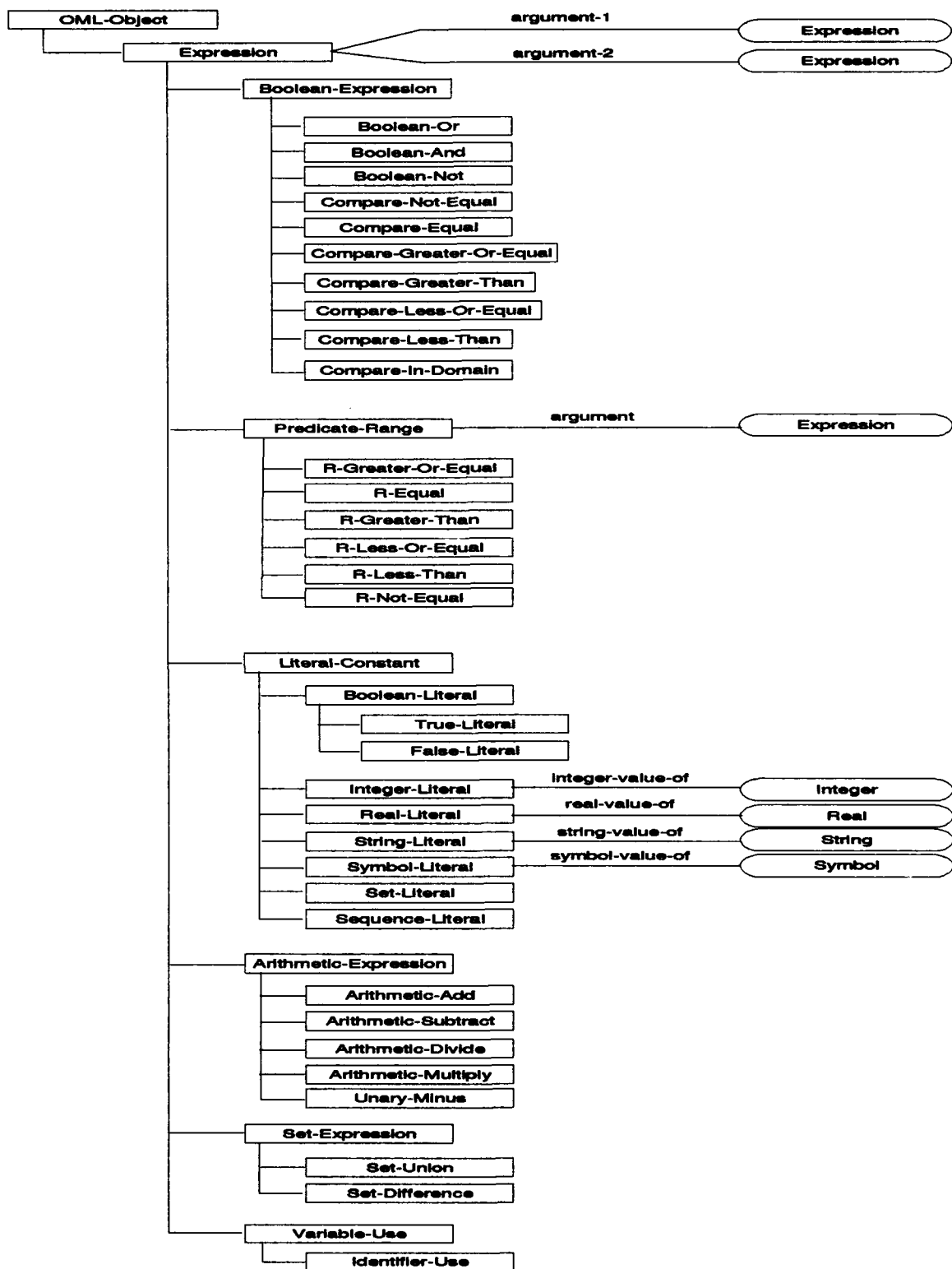


Figure 22. Hierarchy Detail with Object Mappings, Continued

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% File-Name : dm.re (OML domain model)
%%%
%%% Authors : Capt Mary Boom, Capt Brad Mallare
%%%
%%% Purpose : This file builds the domain model to support the OML
%%% architecture defined in Chapter 3 of our thesis. There are three main
%%% types of constructs in this file: Object class definitions, attribute
%%% maps and tree attribute definitions. The object class are defined in
%%% the first part of this file and are written in an ISA type hierarchy.
%%% Attribute maps and tree attribute definitions are partitioned according
%%% to object type.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

!! in-package("RU")
!! in-grammar('user)

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% OBJECT CLASSES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

var OML-Object          : object-class subtype-of user-object
var Ada-Syntactic-Object : object-class subtype-of user-object

var Informal-Model      : object-class subtype-of OML-Object

```

```

%%% The following objects match the objects required in the OML architecture

```

```

var Analysis-Object      : object-class subtype-of OML-Object
var Entity-object        : object-class subtype-of Analysis-Object
var Entity-Class         : object-class subtype-of Entity-Object
var Entity-Instance      : object-class subtype-of Entity-Object
var Process-Object       : object-class subtype-of Analysis-Object
var State-Object         : object-class subtype-of Analysis-Object
var Behavior-Object      : object-class subtype-of Analysis-Object
var Process-Desc-Lang    : object-class subtype-of Behavior-Object
var Pre-Post-Cond        : object-class subtype-of Behavior-Object
var Decision-Table       : object-class subtype-of Behavior-Object
var Store-Object         : object-class subtype-of Analysis-Object
var Relationship-Object   : object-class subtype-of Analysis-Object
var Flow-Object          : object-class subtype-of Analysis-Object
var Event-Object         : object-class subtype-of Analysis-Object
var Relation-Table       : object-class subtype-of Analysis-Object
var Relation             : object-class subtype-of Relation-Table

```

```

%%% The following objects are needed for specifying Behaviors

```

```

var DT-Components        : object-class subtype-of OML-Object
var Condition-Row        : object-class subtype-of DT-Components
var Condition-Entry      : object-class subtype-of DT-Components
var Action-Row           : object-class subtype-of DT-Components
var Action-Entry         : object-class subtype-of DT-Components

var PPC-Components       : object-class subtype-of OML-Object
var PPC-Statement        : object-class subtype-of PPC-Components

```

```

%%% The following are necessary for specifying Entity attributes

```

```

var Entity-Attributes      : object-class subtype-of OML-Object
var   User-Defined-Attr    : object-class subtype-of Entity-Attributes
var   User-Declared-Attr   : object-class subtype-of Entity-Attributes

```

%% The following are necessary for specifying the range of Entity attributes

```

var Attribute-Range        : object-class subtype-of OML-Object
var   Integer-Range        : object-class subtype-of Attribute-Range
var   Real-Range           : object-class subtype-of Attribute-Range
var   Enumerated-Range     : object-class subtype-of Attribute-Range

```

%% The following are Expression objects

```

var Expression             : object-class subtype-of OML-Object

var   Boolean-expression   : object-class subtype-of expression
var   Boolean-And          : object-class subtype-of boolean-expression
var   Boolean-Not          : object-class subtype-of boolean-expression
var   Boolean-Or           : object-class subtype-of boolean-expression
var   Compare-Equal        : object-class subtype-of boolean-expression
var   Compare-Greater-Or-Equal : object-class subtype-of boolean-expression
var   Compare-Greater-Than : object-class subtype-of boolean-expression
var   Compare-Less-Or-Equal : object-class subtype-of boolean-expression
var   Compare-Less-Than    : object-class subtype-of boolean-expression
var   Compare-Not-Equal    : object-class subtype-of boolean-expression
var   Compare-In           : object-class subtype-of boolean-expression
var   Compare-For-All      : object-class subtype-of boolean-expression
var   Compare-Exists       : object-class subtype-of boolean-expression

var   Predicate-Range      : object-class subtype-of expression
var   R-Equal              : object-class subtype-of predicate-range
var   R-Greater-Or-Equal   : object-class subtype-of predicate-range
var   R-Greater-Than       : object-class subtype-of predicate-range
var   R-Less-Or-Equal      : object-class subtype-of predicate-range
var   R-Less-Than         : object-class subtype-of predicate-range
var   R-Not-Equal          : object-class subtype-of predicate-range

var   Arithmetic-expression : object-class subtype-of expression
var   Arithmetic-Add        : object-class subtype-of arithmetic-expression
var   Arithmetic-Subtract   : object-class subtype-of arithmetic-expression
var   Unary-Minus          : object-class subtype-of arithmetic-expression
var   Arithmetic-Divide     : object-class subtype-of arithmetic-expression
var   Arithmetic-Multiply   : object-class subtype-of arithmetic-expression

var   Set-expression       : object-class subtype-of expression
var   Set-Union            : object-class subtype-of set-expression
var   Set-Diff             : object-class subtype-of set-expression
var   GetItem              : object-class subtype-of set-expression
var   GetSet               : object-class subtype-of set-expression
var   SetBuilder           : object-class subtype-of set-expression

var   Literal-Constant     : object-class subtype-of expression
var   Integer-Literal      : object-class subtype-of literal-Constant
var   Real-Literal         : object-class subtype-of literal-Constant
var   Boolean-Literal      : object-class subtype-of literal-Constant

```



```

%% The following are Statement objects

```

%%% These object classes represent the declared variables

%%% These object classes represent the legal variable types

%%%%%%%%%% ANALYSIS-OBJECT-ATTRIBUTES %%%%%%%%%%

152

```
var VARIABLE-TYPE      : map(OML-Object, Var-Type)      = {}
var ANALYSIS-OBJ-MAP   : map(OML-Object, set(Analysis-Object)) = {}
```

```
define-tree-attributes('Informal-Model, {'Defined-Name,  
'Analysis-Obj-Map}))
```

```
var ARGUMENT-1      : map(expression, expression)      = {}
var ARGUMENT-2      : map(expression, expression)      = {}
var ARGUMENT        : map(predicate-range, expression) = {}
var SET-ARG         : map(expression, set(boolean-expression)) = {}
var SET-DIFF-CONDITION : map(set-expression, seq(expression)) = {}
var SETBUILDER-MAP  : map(set-expression, setbuilder)  = {}
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ENTITY-ATTRIBUTES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

form ENTITY-OBJECT-ATTRIBUTES

```
define-tree-attributes('Entity-Instance, {'Defined-Name,  
'Entity-User-Dec1-Map,  
'Name-Use,  
'Entity-User-Def-Map,  
'External-Entity,  
'Constraints-Map})
```

```

var USER-DECL-ATTR-TYPE      : map(User-Declared-Attr, Var-Type)      = {}
var RANGE-ATTRIBUTE           : map(OML-Object, Attribute-Range)        = {}
var ENUM-RANGE-MAP            : map(Enumerated-Range, set(symbol))      = {}
var INTEGER-HIGH-MAP          : map(Integer-Range, Integer)             = {}
var INTEGER-LOW-MAP           : map(Integer-Range, Integer)             = {}
var REAL-HIGH-MAP             : map(Real-Range, Real)                   = {}
var REAL-LOW-MAP              : map(Real-Range, Real)                   = {}
var SYMBOL-VAL                : map(Entity-Attributes, Symbol)         = {}

```

```

                                'User-Decl-Attr-Type,
                                'Range-Attribute,
                                'Value-Map,
                                'Symbol-Val});

define-tree-attributes('User-Defined-Attr, {'Name-Use,
                                'Value-Map,
                                'Symbol-Val});

define-tree-attributes('Enumerated-Range, {'enum-range-map});

define-tree-attributes('Integer-Range, {'integer-low-map,
                                'integer-high-map});

define-tree-attributes('Real-Range, {'real-low-map,
                                'real-high-map})

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX PROCESS-ATTRIBUTES XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

form PROCESS-OBJECT-ATTRIBUTES

    define-tree-attributes('Process-Object, {'Defined-Name,
                                'Constraints-Map})

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX STATE-ATTRIBUTES XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

var STATE-SPACE-MAP      : map(State-Object, set(expression)) = {}

form STATE-OBJECT-ATTRIBUTES

    define-tree-attributes('State-Object, {'Defined-Name,
                                'State-Space-Map,
                                'Constraints-Map})

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX BEHAVIOR-ATTRIBUTES XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

var BEHAVIOR-PPC-MAP      : map(Pre-Post-Cond, seq(PPC-Statement)) = {}

%%var PDL-MAP             : map(Process-Desc-Lang, Ada-Syntactic-Object) = {}

var PPC-PRE-MAP           : map(PPC-Statement, seq(Expression))     = {}
var PPC-POST-MAP          : map(PPC-Statement, seq(Statement))      = {}
var PPC-EVENT-MAP         : map(PPC-Statement, Identifier-Use)      = {}

var DT-COND-ROW-MAP       : map(Decision-Table, seq(Condition-Row)) = {}
var DT-ACTION-ROW-MAP     : map(Decision-Table, seq(Action-Row))    = {}
var DT-EVENT-MAP          : map(Decision-Table, seq(Identifier-Use)) = {}

var CONDITION-ENTRY-MAP   : map(Condition-Row, seq(Condition-Entry)) = {}
var CONDITION-RANGE       : map(Condition-Entry, Predicate-Range)   = {}
var DONT-CARE-VALUE       : map(Condition-entry, Boolean)           = {}
var ACTION-ENTRY-MAP      : map(Action-Row, seq(Action-Entry))      = {}
var ACTION-VALUE          : map(Action-Entry, Literal-Constant)     = {}
var ACTION-EXPR           : map(Action-Entry, Arithmetic-expression) = {}
form BEHAVIOR-OBJECT-ATTRIBUTES

```

```

define-tree-attributes('Pre-Post-Cond, {'Defined-Name,
                                         'Behavior-PPC-Map,
                                         'Constraints-Map});

define-tree-attributes('PPC-Statement, {'PPC-Pre-Map,
                                         'PPC-Post-Map,
                                         'PPC-Event-Map});

% define-tree-attributes('Process-Desc-Lang, {'Defined-Name,
%                                         'PDL-Map,
%                                         'Constraints-Map});

define-tree-attributes('Condition-Row, {'Name-Use, 'condition-entry-map});

define-tree-attributes('Action-Row, {'Name-Use, 'action-entry-map});

define-tree-attributes('Condition-Entry, {'condition-range,
                                         'dont-care-value,
                                         'name-use});

define-tree-attributes('Action-Entry, {'action-value,
                                         'name-use,
                                         'action-expr});

define-tree-attributes('Decision-Table, {'Defined-Name,
                                         'DT-Cond-Row-Map,
                                         'DT-Action-Row-Map,
                                         'DT-Event-Map,
                                         'Constraints-Map})

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
var NATURE-SET-MAP          : map(Store-Object, boolean)          = {}
var CONTENT-NAME           : map(Store-Object, Identifier-Use)   = {}
var KEY-NAME               : map(Store-Object, Identifier-Use)   = {}
var ORDER-SET-MAP          : map(Store-Object, boolean)          = {}

form STORE-OBJECT-ATTRIBUTES
  define-tree-attributes('Store-Object, {'Defined-Name,
                                         'Nature-Set-Map,
                                         'Content-Name,
                                         'Key-Name,
                                         'Order-Set-Map,
                                         'Constraints-Map})

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
var REL-TYPE-MAP           : map(Relationship-Object, symbol)    = {}
var CARDINALITY-MAP        : map(Relationship-Object, cardinality-type) = {}

form RELATIONSHIP-OBJECT-ATTRIBUTES
  define-tree-attributes('Relationship-Object, {'Defined-Name,
                                         'Rel-Type-Map,
                                         'Cardinality-Map,
                                         'Constraints-Map})

```



```
define-tree-attributes('Display, {'name-use, 'display-set})
```

#### A.4 OML Grammar

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% File-Name : gm.re    (OML grammar productions)
%%%
%%% Authors : Capt Mary Boom, Capt Brad Mallare
%%%
%%% Purpose : This file builds the productions that define the grammar
%%%           for an OML specification. When parsing OML specifications, these
%%%           productions ensure that the specifications satisfy the syntax
%%%           requirements defined in the OML BNF.
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

!! in-package("ru")
!! in-grammar('syntax')

```

grammar OML

start-classes informal-model

file-classes informal-model

productions

```

informal-model ::=
    ["specification" defined-name analysis-obj-map + "" ]
    builds informal-model,

```

%% ENTITY OBJECT PRODUCTIONS %%%

```

entity-class ::=
    [defined-name "class-of" "entity"
     "type" ":" ([ "external" !! external-entity ] | "internal")
     {[ "parts" entity-user-decl-map + ";" ]}
     {[ "constraints" constraints-map + ";" ]}]
    builds entity-class,

```

```

entity-instance ::=
    [defined-name "instance-of"
     ( [ "entity"
        "type" ":" ([ "external" !! external-entity ] | "internal")
        {[ "parts" entity-user-decl-map + ";" ]}
      | [name-use
         {[ "values" entity-user-def-map + ";" ]}]
        )
     {[ "constraints" constraints-map + ";" ]}]
    builds entity-instance,

```

%% USER-DECLARED-ATTRIBUTE-PRODUCTIONS %%%

```

user-declared-attr ::=
    [defined-name ":" variable-type
     {[ "range" "{" range-attribute "}" ]}]

```

```

    [{"init-val" (value-map | symbol-val)}] ]
    builds user-declared-attr,

user-defined-attr ::=
    [name-use ":" (value-map | symbol-val)]
    builds user-defined-attr,

enumerated-range ::=
    [enum-range-map + "," ]
    builds enumerated-range,

integer-range ::=
    [integer-low-map "." "." integer-high-map]
    builds integer-range,

real-range ::=
    [real-low-map "." "." real-high-map]
    builds real-range,

XXXXXXXXXXXXXXXXXXXXXXXXX PROCESS OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

process-object ::=
    [defined-name "instance-of" "process"
     [{"constraints" constraints-map + ";"}]]
    builds process-object,

XXXXXXXXXXXXXXXXXXXXXXXXX STATE OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

state-object ::=
    [defined-name "instance-of" "state"
     "state-space" ":" state-space-map + ";"
     [{"constraints" constraints-map + ";"}]]
    builds state-object,

XXXXXXXXXXXXXXXXXXXXXXXXX BEHAVIOR OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

decision-table ::=
    [defined-name "instance-of" "behavior"
     [dt-cond-row-map + ";" ] "-->"
     [{"dt-action-row-map + ";" }]]
     [{"event" "," dt-event-map + "," }
      [{"constraints" constraints-map + ";" }]]
    builds decision-table,

condition-row ::=
    [name-use "," condition-entry-map + "," ]
    builds condition-row,

action-row ::=
    [name-use "," action-entry-map + "," ]
    builds action-row,

% Name use in the next production allows us to have symbols such as 'safe and
% 'unsafe in the decision tables. It is not a pure use of the map name-use.

condition-entry ::=

```



```

[ ("dont-care" !! dont-care-value) | condition-range) ] %%| name-use
builds condition-entry,

action-entry ::=
[ ( action-value | name-use | action-expr) ]
builds action-entry,

pre-post-cond ::=
[defined-name "instance-of" "behavior"
  behavior-ppc-map + ";"
  {"constraints" constraints-map + ";"}]
builds pre-post-cond,

ppc-statement ::=
[ppc-pre-map + "&" "-->" {[ppc-post-map + "&"]}
  "event" ppc-event-map]
builds ppc-statement,

% process-desc-lang ::=
%   [defined-name pdl-map
%     {"constraints" constraints-map + ";"}]
%   builds process-desc-lang,

%% place-holder for pdl, pending implementation

% ada-syntactic-object ::=
%   ["null"] builds ada-syntactic-object,

XXXXXXXXXXXXXXXXXXXXXXXXX STORE OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

store-object ::=
[defined-name "instance-of" "store"
  "nature" ":" (["set" !! nature-set-map] | "sequence")
  "content" ":" content-name
  {"key" ":" key-name}
  {"order" ":" (["ascending" !! order-set-map] | "descending")}
  {"constraints" constraints-map + ";"}]
builds store-object,

XXXXXXXXXXXXXXXXXXXXXXXXX RELATIONSHIP OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

relationship-object ::=
[defined-name "instance-of" "relationship"
  "type" ":" rel-type-map
  "cardinality" ":" cardinality-map
  {"constraints" constraints-map + ";"}]
builds relationship-object,

XXXXXXXXXXXXXXXXXXXXXXXXX FLOW OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

flow-object ::=
[defined-name "instance-of" "flow"
  "flow-link" ":" flow-link-map
  "flow-data" ":" flow-data-map
  {"constraints" constraints-map + ";"}]
builds flow-object,

```

XXXXXXXXXXXXXXXXXXXXXXXXX EVENT OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
event-object ::=
  [defined-name "instance-of" "event"
    "type" ":" ([ "internal" !! event-type ] | "external")
    [{"priority" ":" event-priority}]
    [{"constraints" constraints-map + ";"}]]
  builds event-object,
```

XXXXXXXXXXXXXXXXXXXXXXXXX RELATION-TABLE OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
relation-table ::=
  [defined-name "instance-of" "relation-table"
    relation-sequence + ";"
    [{"constraints" constraints-map + ";"}]]
  builds relation-table,
```

```
relation ::=
  [from-obj-map "," assoc-obj-map "," to-obj-map]
  builds relation,
```

XXXXXXXXXXXXXXXXXXXXXXXXX IDENTIFIER OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
identifier-def ::= [ name ]
  builds identifier-def,

identifier-use ::= [ name ]
  builds identifier-use,
```

XXXXXXXXXXXXXXXXXXXXXXXXX LITERAL CONSTANT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
integer-literal ::= [ integer-value-of ]
  builds integer-literal,

real-literal ::= [ real-value-of ]
  builds real-literal,

true-literal ::= [ "true" ]
  builds true-literal,

false-literal ::= [ "false" ]
  builds false-literal,

string-literal ::= [ string-value-of ]
  builds string-literal,
```

XXXXXXXXXXXXXXXXXXXXXXXXX VARIABLE TYPE PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
type-boolean ::= [ "boolean" ]
  builds type-boolean,

type-string ::= [ "string" ]
  builds type-string,

type-symbol ::= [ "symbol" ]
```

```

        builds type-symbol,

type-integer ::= [ "integer" ]
        builds type-integer,

type-real ::= [ "real" ]
        builds type-real,

type-set ::= [ "set" ]
        builds type-set,

type-sequence ::= [ "sequence" ]
        builds type-sequence,

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXX FLOW TYPE PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX

```

flow-pp ::= [ "proc-proc" ]
        builds flow-pp,

flow-ps ::= [ "proc-store" ]
        builds flow-ps,

flow-sp ::= [ "store-proc" ]
        builds flow-sp,

flow-pe ::= [ "proc-entity" ]
        builds flow-pe,

flow-ep ::= [ "entity-proc" ]
        builds flow-ep,

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXX CARDINALITY TYPE PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX

```

One-Many ::= [ "1-m" ]
        builds One-Many,

One-One ::= [ "1-1" ]
        builds One-One,

Many-One ::= [ "m-1" ]
        builds Many-One,

Many-Many ::= [ "m-m" ]
        builds Many-Many,

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXX EXPRESSION PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX

XXX Arithmetic Expressions XXX

```

arithmetic-add ::= [ argument-1 "+" argument-2 ] builds arithmetic-add,

unary-minus ::= [ "-" argument-1 ] builds unary-minus,

arithmetic-divide ::= [ argument-1 "/" argument-2 ]
        builds arithmetic-divide,

```

```

arithmetic-multiply ::= [ argument-1 "*" argument-2 ]
    builds arithmetic-multiply,

arithmetic-subtract ::= [ argument-1 "-" argument-2 ]
    builds arithmetic-subtract,

%%% Boolean Expressions %%%

boolean-and ::= [ argument-1 "and" argument-2 ]
    builds boolean-and,

boolean-not ::= [ "not" argument-1 ]
    builds boolean-not,

boolean-or ::= [ argument-1 "or" argument-2 ]
    builds boolean-or,

compare-equal ::= [ argument-1 "=" argument-2 ]
    builds compare-equal,

compare-greater-or-equal ::= [ argument-1 ">=" argument-2 ]
    builds compare-greater-or-equal,

compare-greater-than ::= [ argument-1 ">" argument-2 ]
    builds compare-greater-than,

compare-less-or-equal ::= [ argument-1 "<=" argument-2 ]
    builds compare-less-or-equal,

compare-less-than ::= [ argument-1 "<" argument-2 ]
    builds compare-less-than,

compare-not-equal ::= [ argument-1 "/=" argument-2 ]
    builds compare-not-equal,

compare-in ::= [ argument-1 "in" argument-2 ]      %% arg-2 must be a set
    builds compare-in,                          %% or seq

compare-for-all ::= ["forall" "(" name-uses + "," ")"
    "(" [set-arg + "&"] ">" argument-1 ")"]
    builds compare-for-all,

compare-exists ::= ["exists" "(" name-uses + "," ")"
    "(" [set-arg + "&"] ")" ]
    builds compare-exists,

%%% Set comprehension expressions %%%

set-union ::= [argument-1 "union" argument-2]
    builds set-union,

set-diff ::= [argument-1 "set-diff" setbuilder-map ]
    builds set-diff,

getitem ::= [ "getitem" "(" setbuilder-map ")" ]
    builds getitem,

```

```

getset ::= [ "getset" setbuilder-map ]
        builds getset,

setbuilder ::= [ "{" defined-name "|" set-diff-condition + "&" "}" ]
        builds setbuilder,

%%% Predicate Range Expressions %%%

r-equal ::= ["=" argument ]
        builds r-equal,

r-greater-or-equal ::= [ ">=" argument ]
        builds r-greater-or-equal,

r-greater-than ::= [ ">" argument ]
        builds r-greater-than,

r-less-or-equal ::= [ "<=" argument ]
        builds r-less-or-equal,

r-less-than ::= [ "<" argument ]
        builds r-less-than,

r-not-equal ::= [ "/=" argument ]
        builds r-not-equal,

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
assignment-statement ::= [ LHS ":" RHS ]
        builds assignment-statement,

create ::= [ "create" "(" defined-name ":" name-use ")" ]
        builds create,

destroy ::= [ "destroy" "(" name-use ")" ]
        builds destroy,

display ::= [ "display" "(" (name-use | display-set ) ")" ]
        builds display

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
no-patterns

precedence

for expression brackets "(" matching ")"

(same-level "and", "or" associativity left),
(same-level "<", "<=", "=", ">=", ">", "/=" associativity none),
(same-level "in", "set-diff", "union" associativity left),
(same-level "+", "-" associativity left),
(same-level "*", "/" associativity left),
(same-level "not" associativity none)

symbol-start-chars

```

```
"abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ"

symbol-continue-chars
"abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-."

comments
    "%" matching "
",
    "#|" matching "||#" nested

end
```

## *Appendix B. Ada (Subset) Program Design Language (PDL)*

### 1. Program Parts

<Ada-Program> ::= <Procedure-Declaration>

<Procedure-Declaration> ::= **"procedure"** <Identifier-Definition> **"is"** <Ablock>

<Function-Declaration> ::= **"function"** <Identifier-Definition>

**"return"** <Var-Type> **"is"** <Ablock>

<Ablock> ::= {<Declaration-Statement>} **"begin"**

{<Astatement>}+ **"end"** {<Identifier-Definition>} **";"**

<Declaration-Statement> ::= <Variable-Declaration>

| <Enumerated-Declaration> | <Subprogram-Declaration>

<Variable-Declaration> ::= <Identifier-Definition> {, <Identifier-Definition>} **":"**

[<Constant-Flag>] <Var-Type> [**":"**<Expression>] **";"**

<Enumerated-Declaration> ::= **"type"** <Identifier-Definition> **"is"**

**"("** <Identifier-Definition> {, <Identifier-Definition>} **")"** **";"**

<Subprogram-Declaration> ::= (<Procedure-Declaration> | <Function-Declaration>)

<Var-Type> ::= **"boolean"** | **"integer"** | **"float"** | <Type-Enumerated> | **"string"**

<Type-Enumerated> ::= <Identifier-Use>

### 2. Statements

<Astatement> ::= <An-If-Statement> | <Assignment-Statement>

| <Loop-Statement> | <Exit-Statement> | <Return-Statement>

| <Read-Statement> | <Write-Statement>

| <Procedure-Call> | <Block-Structure>

$\langle \text{An-If-Statement} \rangle ::= \text{"if"} \langle \text{Expression} \rangle \text{"then"} \{ \langle \text{Astatement} \rangle \} +$   
 $\quad \{ \{ \langle \text{Elsif-Statement} \rangle \} [ \text{"else"} \{ \langle \text{Astatement} \rangle \} + ] ] \text{"end"} \text{"if"} \text{";"}$

$\langle \text{Elsif-Statement} \rangle ::= \text{"elsif"} \langle \text{Expression} \rangle \text{"then"} \{ \langle \text{Astatement} \rangle \} +$

$\langle \text{Assignment-Statement} \rangle ::= \langle \text{Variable-Use} \rangle \text{"="} \langle \text{Expression} \rangle \text{";"}$

$\langle \text{Loop-Statement} \rangle ::= \langle \text{Basic-Loop} \rangle \mid \langle \text{For-Loop} \rangle \mid \langle \text{While-Loop} \rangle$

$\langle \text{Basic-Loop} \rangle ::= [ \langle \text{Identifier-Definition} \rangle \text{":"} ]$   
 $\quad \text{"loop"} \{ \langle \text{Astatement} \rangle \} + \text{"end"} \text{"loop"} \text{";"}$

$\langle \text{For-Loop} \rangle ::= [ \langle \text{Identifier-Definition} \rangle \text{":"} ]$   
 $\quad \text{"for"} \langle \text{Identifier-Use} \rangle \text{"in"} \langle \text{Expression} \rangle \text{"."} \text{"."} \langle \text{Expression} \rangle$   
 $\quad \text{"loop"} \{ \langle \text{Astatement} \rangle \} + \text{"end"} \text{"loop"} \text{";"}$

$\langle \text{While-Loop} \rangle ::= [ \langle \text{Identifier-Definition} \rangle \text{":"} ] \text{"while"}$   
 $\quad \langle \text{Boolean-Expression} \rangle \text{"loop"} \{ \langle \text{Astatement} \rangle \} + \text{"end"} \text{"loop"} \text{";"}$

$\langle \text{ExitStatement} \rangle ::= \text{"exit"} [ \langle \text{Identifier-Definition} \rangle ] [ \text{"when"} \langle \text{Expression} \rangle ] \text{";"}$

$\langle \text{Read-Statement} \rangle ::= \text{"read"} ( \langle \text{Variable-Use} \rangle ) \text{";"}$

$\langle \text{Write-Statement} \rangle ::= \text{"write"} ( \langle \text{Expression} \rangle ) \text{";"}$

$\langle \text{Return-Statement} \rangle ::= \text{"return"} \langle \text{Expression} \rangle \text{";"}$

$\langle \text{Block-Structure} \rangle ::= [ \langle \text{Identifier-Definition} \rangle \text{":"} ] \{ \text{"declare"} \langle \text{Declaration-Statement} \rangle \}$   
 $\quad \text{"begin"} \{ \langle \text{Astatement} \rangle \} + \text{"end"} [ \langle \text{Identifier-Definition} \rangle ] \text{";"}$

$\langle \text{Procedure-Call} \rangle ::= \langle \text{Identifier-Use} \rangle ( ) \text{";"}$

### 3. Expressions

$\langle \text{Expression} \rangle ::= \langle \text{Boolean-Expression} \rangle \mid \langle \text{Arithmetic-Expression} \rangle$   
 $\quad \mid \langle \text{Afunction-Call} \rangle \mid \langle \text{Variable-Use} \rangle \mid \langle \text{Type-Conversion-Expression} \rangle$   
 $\quad \mid \langle \text{Literal-Constant} \rangle \mid \langle \text{Emuneration-Expression} \rangle$



$\langle \text{Boolean-Expression} \rangle ::= \langle \text{Boolean-And} \rangle \mid \langle \text{Boolean-Or} \rangle$   
 $\mid \langle \text{Boolean-Not} \rangle \mid \langle \text{Compare-Equal} \rangle \mid \langle \text{Compare-Not-Equal} \rangle$   
 $\mid \langle \text{Compare-Greater-Or-Equal} \rangle \mid \langle \text{Compare-Greater-Than} \rangle$   
 $\mid \langle \text{Compare-Less-Or-Equal} \rangle \mid \langle \text{Compare-Less-Than} \rangle$

$\langle \text{Boolean-And} \rangle ::= \langle \text{Argument-1} \rangle \text{ and } \langle \text{Argument-2} \rangle$

$\langle \text{Boolean-Or} \rangle ::= \langle \text{Argument-1} \rangle \text{ or } \langle \text{Argument-2} \rangle$

$\langle \text{Boolean-Not} \rangle ::= \text{not " } \langle \text{Argument-1} \rangle \text{ "}$

$\langle \text{Compare-Equal} \rangle ::= \langle \text{Argument-1} \rangle \text{ "=" } \langle \text{Argument-2} \rangle$

$\langle \text{Compare-Greater-Or-Equal} \rangle ::= \langle \text{Argument-1} \rangle \text{ ">=" } \langle \text{Argument-2} \rangle$

$\langle \text{Compare-Greater-Than} \rangle ::= \langle \text{Argument-1} \rangle \text{ ">" } \langle \text{Argument-2} \rangle$

$\langle \text{Compare-Less-Or-Equal} \rangle ::= \langle \text{Argument-1} \rangle \text{ "<=" } \langle \text{Argument-2} \rangle$

$\langle \text{Compare-Less-Than} \rangle ::= \langle \text{Argument-1} \rangle \text{ "<" } \langle \text{Argument-2} \rangle$

$\langle \text{Compare-Not-Equal} \rangle ::= \langle \text{Argument-1} \rangle \text{ "/=" } \langle \text{Argument-2} \rangle$

$\langle \text{Arithmetic-Expression} \rangle ::= \langle \text{Arithmetic-Add} \rangle \mid \langle \text{Arithmetic-Subtract} \rangle$   
 $\mid \langle \text{Arithmetic-Divide} \rangle \mid \langle \text{Arithmetic-Modulo} \rangle$   
 $\mid \langle \text{Arithmetic-Multiply} \rangle \mid \langle \text{Arithmetic-Abs} \rangle$   
 $\mid \langle \text{Arithmetic-Exponent} \rangle \mid \langle \text{Unary-Plus} \rangle \mid \langle \text{Unary-Minus} \rangle$

$\langle \text{Arithmetic-Add} \rangle ::= \langle \text{Argument-1} \rangle \text{ "+" } \langle \text{Argument-2} \rangle$

$\langle \text{Unary-Plus} \rangle ::= \text{"+" } \langle \text{Argument-1} \rangle$

$\langle \text{Unary-Minus} \rangle ::= \text{"-" } \langle \text{Argument-1} \rangle$

$\langle \text{Arithmetic-Divide} \rangle ::= \langle \text{Argument-1} \rangle \text{ "/" } \langle \text{Argument-2} \rangle$

$\langle \text{Arithmetic-Modulo} \rangle ::= \langle \text{Argument-1} \rangle \text{ mod } \langle \text{Argument-2} \rangle$

$\langle \text{Arithmetic-Multiply} \rangle ::= \langle \text{Argument-1} \rangle \text{ "*" } \langle \text{Argument-2} \rangle$

**<Arithmetic-Subtract> ::= <Argument-1> "-" <Argument-2>**

**<Arithmetic-Abs> ::= **abs** "(" <Argument-1> ")"**

**<Arithmetic-Exponent> ::= <Argument-1> **\*\*** <Argument-2>**

**<Afunction-Call> ::= <Identifier-Use> "("**

**<Variable-Use> ::= <Identifier-Use>**

**<Literal-Constant> ::= <Integer-Literal> | <Real-Literal> | <Boolean-Literal>**

**<Literal-Constant> ::= <False-Literal> | <True-Literal>**

**<Enumeration-Expression> ::= <Succ-Expression> | <Pred-Expression>  
| <Char-Expression> | <Val-Expression>**

**<Succ-Expression> ::= <Identifier-Use> **"succ"** "(" <Identifier-Use> ")"**

**<Pred-Expression> ::= <Identifier-Use> **"pred"** "(" <Identifier-Use> ")"**

**<Char-Expression> ::= <Identifier-Use> **"char"****

**<Val-Expression> ::= <Identifier-Use> **"val"** "(" <Expression> ")"**

**<Type-Conversion-Expression> ::= <Int-To-Float> | <Float-To-Int>**

**<Float-To-Int> ::= **"integer"** "(" <Expression> ")"**

**<Int-To-Float> ::= **"float"** "(" <Expression> ")"**

#### 4. Literals and Identifiers

**<Argument-1> ::= <Expression>**

**<Argument-2> ::= <Expression>**

**<Identifier-Definition> ::= <Name>**

**<Identifier-Use> ::= <Name>**

**<Integer-Literal> ::= integer**

**<Real-Literal> ::= real**

**<True-Literal> ::= "true"**

**<False-Literal> ::= "false"**

**<Constant-Flag> ::= "constant"**

**<Name> ::= symbol**

## B.1 OML with Ada PDL Domain Model

### B.1.1 OML Domain Model

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% File-Name : dm.re (OML domain model)
%%%
%%% Authors : Capt Mary Boom, Capt Brad Mallare
%%%
%%% Purpose : This file builds the domain model to support the OML
%%% architecture defined in Chapter 3 of our thesis. There are three main
%%% types of constructs in this file: Object class definitions, attribute
%%% maps and tree attribute definitions. The object class are defined in
%%% the first part of this file and are written in an ISA type hierarchy.
%%% Attribute maps and tree attribute definitions are partitioned according
%%% to object type.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
!! in-package("RU")
!! in-grammar('user')
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% OBJECT CLASSES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
var Specification-Object      : object-class subtype-of user-object
var OML-Object                : object-class subtype-of specification-object
```

```
var Informal-Model           : object-class subtype-of OML-Object
```

```
%%% The following objects match the objects required in the OML architecture
```

```
var Analysis-Object          : object-class subtype-of OML-Object
var Entity-object            : object-class subtype-of Analysis-Object
var Entity-Class              : object-class subtype-of Entity-Object
var Entity-Instance          : object-class subtype-of Entity-Object
var Process-Object            : object-class subtype-of Analysis-Object
var State-Object              : object-class subtype-of Analysis-Object
var Behavior-Object           : object-class subtype-of Analysis-Object
var Process-Desc-Lang         : object-class subtype-of Behavior-Object
var Pre-Post-Cond             : object-class subtype-of Behavior-Object
var Decision-Table            : object-class subtype-of Behavior-Object
var Store-Object              : object-class subtype-of Analysis-Object
var Relationship-Object        : object-class subtype-of Analysis-Object
var Flow-Object               : object-class subtype-of Analysis-Object
var Event-Object              : object-class subtype-of Analysis-Object
var Relation-Table            : object-class subtype-of Analysis-Object
var Relation                  : object-class subtype-of Relation-Table
```

```
%%% The following objects are needed for specifying Behaviors
```

```
var DT-Components            : object-class subtype-of OML-Object
var Condition-Row             : object-class subtype-of DT-Components
var Condition-Entry           : object-class subtype-of DT-Components
var Action-Row                : object-class subtype-of DT-Components
var Action-Entry              : object-class subtype-of DT-Components
```

```
var PPC-Components           : object-class subtype-of OML-Object
```

```

var    PPC-Statement          : object-class subtype-of PPC-Components

var    %% The following are necessary for specifying Entity attributes

    Entity-Attributes        : object-class subtype-of OML-Object
var    User-Defined-Attr     : object-class subtype-of Entity-Attributes
var    User-Declared-Attr    : object-class subtype-of Entity-Attributes

%% The following are necessary for specifying the range of Entity attributes

var    Attribute-Range       : object-class subtype-of OML-Object
var    Integer-Range         : object-class subtype-of Attribute-Range
var    Real-Range            : object-class subtype-of Attribute-Range
var    Enumerated-Range      : object-class subtype-of Attribute-Range

%% The following are Expression objects

var    Expression            : object-class subtype-of specification-Object

var    Boolean-expression    : object-class subtype-of expression
var    Boolean-And           : object-class subtype-of boolean-expression
var    Boolean-Not           : object-class subtype-of boolean-expression
var    Boolean-Or            : object-class subtype-of boolean-expression
var    Compare-Equal         : object-class subtype-of boolean-expression
var    Compare-Greater-Or-Equal : object-class subtype-of boolean-expression
var    Compare-Greater-Than  : object-class subtype-of boolean-expression
var    Compare-Less-Or-Equal : object-class subtype-of boolean-expression
var    Compare-Less-Than     : object-class subtype-of boolean-expression
var    Compare-Not-Equal     : object-class subtype-of boolean-expression
var    Compare-In            : object-class subtype-of boolean-expression
var    Compare-For-All       : object-class subtype-of boolean-expression
var    Compare-Exists        : object-class subtype-of boolean-expression

var    Predicate-Range      : object-class subtype-of expression
var    R-Equal              : object-class subtype-of predicate-range
var    R-Greater-Or-Equal   : object-class subtype-of predicate-range
var    R-Greater-Than       : object-class subtype-of predicate-range
var    R-Less-Or-Equal      : object-class subtype-of predicate-range
var    R-Less-Than          : object-class subtype-of predicate-range
var    R-Not-Equal          : object-class subtype-of predicate-range

var    Arithmetic-expression : object-class subtype-of expression
var    Arithmetic-Add        : object-class subtype-of arithmetic-expression
var    Arithmetic-Subtract   : object-class subtype-of arithmetic-expression
var    Unary-Minus           : object-class subtype-of arithmetic-expression
var    Arithmetic-Divide     : object-class subtype-of arithmetic-expression
var    Arithmetic-Multiply   : object-class subtype-of arithmetic-expression

var    Set-expression        : object-class subtype-of expression
var    Set-Union             : object-class subtype-of set-expression
var    Set-Diff              : object-class subtype-of set-expression
var    GetItem               : object-class subtype-of set-expression
var    GetSet                : object-class subtype-of set-expression
var    SetBuilder            : object-class subtype-of set-expression

var    Literal-Constant      : object-class subtype-of expression

```





form USER-DECLARED-ATTRIBUTES

```
define-tree-attributes('User-Declared-Attr, {'Defined-Name,
                                             'User-Decl-Attr-Type,
                                             'Range-Attribute,
                                             'Value-Map,
                                             'Symbol-Val});
```

```
define-tree-attributes('User-Defined-Attr, {'Name-Use,
                                             'Value-Map,
                                             'Symbol-Val});
```

```
define-tree-attributes('Enumerated-Range, {'enum-range-map});
```

```
define-tree-attributes('Integer-Range, {'integer-low-map,
                                         'integer-high-map});
```

```
define-tree-attributes('Real-Range, {'real-low-map,
                                     'real-high-map});
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX PROCESS-ATTRIBUTES XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

form PROCESS-OBJECT-ATTRIBUTES

```
define-tree-attributes('Process-Object, {'Defined-Name,
                                          'Constraints-Map});
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX STATE-ATTRIBUTES XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
var STATE-SPACE-MAP : map(State-Object, set(expression)) = {}
```

form STATE-OBJECT-ATTRIBUTES

```
define-tree-attributes('State-Object, {'Defined-Name,
                                         'State-Space-Map,
                                         'Constraints-Map});
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX BEHAVIOR-ATTRIBUTES XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```
var BEHAVIOR-PPC-MAP : map(Pre-Post-Cond, seq(PPC-Statement)) = {}
```

```
var PDL-MAP : map(Process-Desc-Lang, Procedure-Declaration) = {}
```

```
var PPC-PRE-MAP : map(PPC-Statement, seq(Expression)) = {}
```

```
var PPC-POST-MAP : map(PPC-Statement, seq(Statement)) = {}
```

```
var PPC-EVENT-MAP : map(PPC-Statement, Identifier-Use) = {}
```

```
var DT-COND-ROW-MAP : map(Decision-Table, seq(Condition-Row)) = {}
```

```
var DT-ACTION-ROW-MAP : map(Decision-Table, seq(Action-Row)) = {}
```

```
var DT-EVENT-MAP : map(Decision-Table, seq(Identifier-Use)) = {}
```

```
var CONDITION-ENTRY-MAP : map(Condition-Row, seq(Condition-Entry)) = {}
```

```
var CONDITION-RANGE : map(Condition-Entry, Predicate-Range) = {}
```

```
%var CONDITION-VALUE : map(Condition-Entry, Literal-Constant) = {}
```



```

var DONT-CARE-VALUE      : map(Condition-entry, Boolean)          = {}
var ACTION-ENTRY-MAP     : map(Action-Row, seq(Action-Entry))     = {}
var ACTION-VALUE         : map(Action-Entry, Literal-Constant)    = {}
var ACTION-EXPR          : map(Action-Entry, Arithmetic-expression) = {}
form BEHAVIOR-OBJECT-ATTRIBUTES

```

```

define-tree-attributes('Pre-Post-Cond, {'Defined-Name,
                                         'Behavior-PPC-Map,
                                         'Constraints-Map});

```

```

define-tree-attributes('PPC-Statement, {'PPC-Pre-Map,
                                         'PPC-Post-Map,
                                         'PPC-Event-Map});

```

```

define-tree-attributes('Process-Desc-Lang, {'Defined-Name,
                                             'PDL-Map,
                                             'Constraints-Map});

```

```

define-tree-attributes('Condition-Row, {'Name-Use, 'condition-entry-map});

```

```

define-tree-attributes('Action-Row, {'Name-Use, 'action-entry-map});

```

```

define-tree-attributes('Condition-Entry, {'condition-range,
                                           'dont-care-value,
                                           'name-use});

```

```

define-tree-attributes('Action-Entry, {'action-value,
                                       'name-use,
                                       'action-expr});

```

```

define-tree-attributes('Decision-Table, {'Defined-Name,
                                          'DT-Cond-Row-Map,
                                          'DT-Action-Row-Map,
                                          'DT-Event-Map,
                                          'Constraints-Map});

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX STORE-ATTRIBUTES XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

var NATURE-SET-MAP       : map(Store-Object, boolean)            = {}
var CONTENT-NAME         : map(Store-Object, Identifier-Use)     = {}
var KEY-NAME             : map(Store-Object, Identifier-Use)     = {}
var ORDER-SET-MAP        : map(Store-Object, boolean)            = {}

```

form STORE-OBJECT-ATTRIBUTES

```

define-tree-attributes('Store-Object, {'Defined-Name,
                                         'Nature-Set-Map,
                                         'Content-Name,
                                         'Key-Name,
                                         'Order-Set-Map,
                                         'Constraints-Map});

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX RELATIONSHIP-ATTRIBUTES XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

var REL-TYPE-MAP         : map(Relationship-Object, symbol)      = {}
var CARDINALITY-MAP      : map(Relationship-Object, cardinality-type) = {}

```

```

form RELATIONSHIP-OBJECT-ATTRIBUTES
  define-tree-attributes('Relationship-Object, {'Defined-Name,
    'Rel-Type-Map,
    'Cardinality-Map,
    'Constraints-Map})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FLOW-ATTRIBUTES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var FLOW-LINK-MAP      : map(Flow-Object, flow-type)      = {}
var FLOW-DATA-MAP      : map(Flow-Object, identifier-use) = {}

form FLOW-OBJECT-ATTRIBUTES
  define-tree-attributes('Flow-Object, {'Defined-Name,
    'Flow-Link-Map,
    'Flow-Data-Map,
    'Constraints-Map})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% EVENT-ATTRIBUTES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var EVENT-TYPE          : map(Event-Object, boolean)      = {}
var EVENT-PRIORITY      : map(Event-Object, integer)      = {}

form EVENT-OBJECT-ATTRIBUTES
  define-tree-attributes('Event-Object, {'Defined-Name,
    'Event-Type,
    'Event-Priority,
    'Constraints-Map})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% RELATION-TABLE-ATTRIBUTES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var RELATION-SEQUENCE   : map(Relation-Table, seq(Relation)) = {}
var FROM-OBJ-MAP        : map(Relation, Identifier-Use)      = {}
var ASSOC-OBJ-MAP       : map(Relation, Identifier-Use)      = {}
var TO-OBJ-MAP          : map(Relation, Identifier-Use)      = {}

form RELATION-TABLE-ATTRIBUTES
  define-tree-attributes('Relation-Table, {'Defined-Name,
    'Relation-Sequence,
    'Constraints-Map});

  define-tree-attributes('Relation, {'From-Obj-Map,
    'Assoc-Obj-Map,
    'To-Obj-Map})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ASSIGNMENT-STATEMENTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var LHS      : map(Assignment-Statement, variable-use) = {}
var RHS      : map(Assignment-Statement, expression)  = {}

form ASSIGNMENT-STATEMENT-ATTRIBUTES
  define-tree-attributes('Assignment-Statement, {'LHS, 'RHS})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FUNCTION-CALLS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var DISPLAY-SET      : map(display, set-expression)      = {}

```

form FUNCTION-CALL-ATTRIBUTES

define-tree-attributes('Create, {'defined-name, 'name-use});

define-tree-attributes('Destroy, {'name-use});

define-tree-attributes('Display, {'name-use, 'display-set})

### B.1.2 Ada PDL Domain Model

```
!! in-package("ru")
!! in-grammar('user)

var Specification-Object      : object-class subtype-of user-object

var Ada-Syntactic-Object      : object-class subtype-of specification-object

var Procedure-declaration     : object-class subtype-of ada-syntactic-object
var Function-declaration      : object-class subtype-of ada-syntactic-object
var ABlock                    : object-class subtype-of ada-syntactic-object
var Declaration-statement     : object-class subtype-of ada-syntactic-object
var Variable-declaration      : object-class subtype-of declaration-statement
var Enumerated-declaration    : object-class subtype-of declaration-statement
var Subprogram-declaration    : object-class subtype-of declaration-statement
var Var-Type                   : object-class subtype-of specification-object
var Type-boolean               : object-class subtype-of var-type
var Type-integer               : object-class subtype-of var-type
var Type-float                 : object-class subtype-of var-type
var Type-enumerated            : object-class subtype-of var-type
var Type-string                : object-class subtype-of var-type

var Constant-flag              : object-class subtype-of ada-syntactic-object

var Expression                 : object-class subtype-of specification-object

var Boolean-expression         : object-class subtype-of expression
var Boolean-And                 : object-class subtype-of boolean-expression
var Boolean-Not                 : object-class subtype-of boolean-expression
var Boolean-Or                  : object-class subtype-of boolean-expression
var Compare-Equal               : object-class subtype-of boolean-expression
var Compare-Greater-Or-Equal    : object-class subtype-of boolean-expression
var Compare-Greater-Than       : object-class subtype-of boolean-expression
var Compare-Less-Or-Equal       : object-class subtype-of boolean-expression
var Compare-Less-Than          : object-class subtype-of boolean-expression
var Compare-Not-Equal           : object-class subtype-of boolean-expression

var Arithmetic-expression      : object-class subtype-of expression
var Arithmetic-Add              : object-class subtype-of arithmetic-expression
var Arithmetic-Subtract        : object-class subtype-of arithmetic-expression
var Arithmetic-Divide          : object-class subtype-of arithmetic-expression
var Arithmetic-Modulo          : object-class subtype-of arithmetic-expression
var Arithmetic-Multiply        : object-class subtype-of arithmetic-expression
var Arithmetic-Abs              : object-class subtype-of arithmetic-expression
var Unary-Plus                  : object-class subtype-of arithmetic-expression
var Unary-Minus                 : object-class subtype-of arithmetic-expression
var Arithmetic-Exponent        : object-class subtype-of arithmetic-expression

var AFunction-call             : object-class subtype-of expression
var Variable-Use                : object-class subtype-of expression
var Identifier-Use              : object-class subtype-of Variable-Use

var Literal-Constant           : object-class subtype-of expression
var Integer-Literal            : object-class subtype-of literal-Constant
var Real-Literal                : object-class subtype-of literal-Constant
```

```

var      Boolean-Literal      : object-class subtype-of literal-Constant
var      False-Literal       : object-class subtype-of Boolean-Literal
var      True-Literal        : object-class subtype-of Boolean-Literal

var      Enumeration-expression : object-class subtype-of expression
var      Succ-expression      : object-class subtype-of enumeration-expression
var      Pred-expression      : object-class subtype-of enumeration-expression
var      Char-expression      : object-class subtype-of enumeration-expression
var      Val-expression       : object-class subtype-of enumeration-expression

var      Type-conversion-expression : object-class subtype-of expression
var      int-to-float         : object-class subtype-of Type-conversion-expression
var      float-to-int        : object-class subtype-of Type-conversion-expression

var      AStatement           : object-class subtype-of ada-syntactic-object
var      An-If-Statement      : object-class subtype-of astatement
var      Elself-statement     : object-class subtype-of ada-syntactic-object
var      AAssignment-statement : object-class subtype-of astatement
var      Loop-statement       : object-class subtype-of astatement
var      Basic-Loop           : object-class subtype-of loop-statement
var      For-Loop             : object-class subtype-of loop-statement
var      While-Loop           : object-class subtype-of loop-statement
var      Exit-statement       : object-class subtype-of astatement
var      Read-statement       : object-class subtype-of astatement
var      Write-statement      : object-class subtype-of astatement
var      Return-statement     : object-class subtype-of astatement
var      Block-structure      : object-class subtype-of astatement
var      Procedure-Call       : object-class subtype-of astatement

var      AIdentifier          : object-class subtype-of ada-syntactic-object
var      Identifier-Definition : object-class subtype-of aidentifier

XXXXXXXXXXXXXXXXXXXXXXXXX ADA-SYNTACTIC-OBJECT-ATTRIBUTES XXXXXXXXXXXXXXXXXXXXXXXX

var ADEFINED-NAME      : map(ada-syntactic-object, identifier-definition) = {}
var BDEFINED-NAME      : map(specification-object, identifier-definition) = {}
var U-NAME             : map(identifier-definition, string) = {}
var ADEFINED-NAMES     : map(ada-syntactic-object, seq(identifier-definition))
                        = {}
var EXPRESSION-VALUE   : map(specification-object, expression) = {}
var EXPRESSION-LIST    : map(ada-syntactic-object, seq(expression)) = {}

form U-NAME-TREE-ATTRIBUTES
  define-tree-attributes('identifier-use, {'u-name})

XXXXXXXXXXXXXXXXXXXXXXXXX PROCEDURE AND FUNCTION-DECLARATION XXXXXXXXXXXXXXXXXXXXXXXX

var BLOCK-MAP          : map(ada-syntactic-object, ablock) = {}
var AVARIABLE-TYPE     : map(ada-syntactic-object, var-type) = {}

form PROCEDURE-DECLARATION-TREE-ATTRIBUTES
  define-tree-attributes('procedure-declaration, {'adefined-name,
                        'block-map});

```

```

define-tree-attributes('function-declaration, {'adefined-name,
                                              'avariable-type,
                                              'block-map})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% BLOCKS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var DECLARATION-PART      : map(ada-syntactic-object,
                               seq(declaration-statement)) = {}

var STATEMENT-SEQUENCE   : map(ada-syntactic-object, seq(Astatement)) = {}

form ABLOCK-TREE-ATTRIBUTES
  define-tree-attributes('ablock, {'declaration-part,
                                'statement-sequence,
                                'bdefined-name})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% DECLARATION-STATEMENTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var VARIABLE-DECLARATION-MAP : map(declaration-statement,
                                   variable-declaration)           = {}
var ENUMERATED-DECLARATION-MAP : map(declaration-statement,
                                   enumerated-declaration)          = {}
var SUBPROGRAM-DECLARATION-MAP : map(declaration-statement,
                                   subprogram-declaration)          = {}
var SUBPROG2PROC-DECL        : map(subprogram-declaration,
                                   procedure-declaration)           = {}
var SUBPROG2FUNC-DECL        : map(subprogram-declaration,
                                   function-declaration)             = {}
var CONSTANT-OPTION          : map(variable-declaration, constant-flag) = {}
var ENUMERATED-SYMBOL-MAP     : map(type-enumerated, identifier-use)  = {}

form DECLARATION-STATEMENT-TREE-ATTRIBUTES
  define-tree-attributes('declaration-statement, {'variable-declaration-map,
                                                  'enumerated-declaration-map,
                                                  'subprogram-declaration-map});

  define-tree-attributes('variable-declaration, {'adefined-name,
                                                  'constant-option,
                                                  'avariable-type,
                                                  'expression-value});

  define-tree-attributes('enumerated-declaration, {'adefined-name,
                                                  'adefined-names});

  define-tree-attributes('subprogram-declaration, {'subprog2proc-decl,
                                                  'subprog2func-decl})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% EXPRESSIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var ARGUMENT-1: map(expression, expression) = {}
var ARGUMENT-2: map(expression, expression) = {}
var BOOLEAN-EXPRESSION-VALUE: map(ada-syntactic-object, boolean-expression)
                                = {}

%%% -----

```

%%% Boolean expressions

%%% -----

form BOOLEAN-TREE-ATTRIBUTES

```
define-tree-attributes('boolean-and, {'argument-1, 'argument-2});
define-tree-attributes('boolean-not, {'argument-1});
define-tree-attributes('boolean-or, {'argument-1, 'argument-2});
define-tree-attributes('compare-equal, {'argument-1, 'argument-2});
define-tree-attributes('compare-greater-or-equal, {'argument-1,
                                                    'argument-2});
define-tree-attributes('compare-greater-than, {'argument-1, 'argument-2});
define-tree-attributes('compare-less-or-equal, {'argument-1, 'argument-2});
define-tree-attributes('compare-less-than, {'argument-1, 'argument-2});
define-tree-attributes('compare-not-equal, {'argument-1, 'argument-2})
```

%%% -----

%%% Arithmetic expressions

%%% -----

form ARITHMETIC-TREE-ATTRIBUTES

```
define-tree-attributes('unary-plus, {'argument-1});
define-tree-attributes('unary-minus, {'argument-1});
define-tree-attributes('arithmetic-add, {'argument-1, 'argument-2});
define-tree-attributes('arithmetic-divide, {'argument-1, 'argument-2});
define-tree-attributes('arithmetic-modulo, {'argument-1, 'argument-2});
define-tree-attributes('arithmetic-multiply, {'argument-1, 'argument-2});
define-tree-attributes('arithmetic-subtract, {'argument-1, 'argument-2});
define-tree-attributes('arithmetic-abs, {'argument-1});
define-tree-attributes('arithmetic-exponent, {'argument-1, 'argument-2})
```

%%% -----

%%% Function-call

%%% -----

var ANAME-USE : map(specification-object, identifier-use) = {}

form FUNCTION-CALL-TREE-ATTRIBUTES

```
define-tree-attributes('afunction-call, {'aname-use})
```

%%% -----

%%% Literal-Integer

%%% -----

var INTEGER-VALUE-OF : map(integer-literal, integer) = {}

%%% -----

%%% Literal-Float

%%% -----

var REAL-VALUE-OF : map(real-literal, real) = {}

%%% -----

%%% Enumeration-expressions

%%% -----

```

var ENUMTYPE-NAME      : map(Enumeration-expression, identifier-use) = {}

form ENUMERATION-EXPRESSION-TREE-ATTRIBUTES
  define-tree-attributes('succ-expression, {'enumtype-name, 'aname-use});
  define-tree-attributes('pred-expression, {'enumtype-name, 'aname-use});
  define-tree-attributes('val-expression, {'enumtype-name, 'expression-value});
  define-tree-attributes('char-expression, {'aname-use})

%%% -----
%%% Type-Conversion-Expressions
%%% -----

form TYPE-CONVERSION-TREE-ATTRIBUTES
  define-tree-attributes('float-to-int, {'expression-value});
  define-tree-attributes('int-to-float, {'expression-value})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% STATEMENTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

var STATEMENT-BODY      : map(astatement, astatement) = {}

%%% -----
%%% If-Statement
%%% -----

var THEN-PART           : map(an-if-statement, seq(astatement)) = {}
var ELSIF-PART          : map(an-if-statement, seq(elsif-statement)) = {}
var ELSE-PART           : map(an-if-statement, seq(astatement)) = {}
var ELSIF-STATEMENTS    : map(elsif-statement, seq(astatement)) = {}
var TEST-CONDITION      : map(an-if-statement, expression) = {}
var ELSIF-TEST-CONDITION : map(elsif-statement, expression) = {}

form IF-STATEMENT-TREE-ATTRIBUTES
  define-tree-attributes('an-if-statement, {'test-condition,
                                         'then-part,
                                         'elsif-part,
                                         'else-part });

  define-tree-attributes('elsif-statement, {'elsif-test-condition,
                                         'elsif-statements})

%%% -----
%%% Assignment-Statement
%%% -----

var ALHS                : map(aassignment-statement, variable-use) = {}

form STATEMENT-TREE-ATTRIBUTES
  define-tree-attributes('aassignment-statement, {'alhs, 'expression-value})

%%% -----
%%% Loop-Statements

```



```

%%%-----
var LOOP-ID           : map(Loop-statement, identifier-definition) = {}
var EXIT-MAP          : map(basic-loop, exit-statement)           = {}
var LOOP-STATEMENT-SEQUENCE : map(loop-statement, seq(astatement)) = {}
var LOOP-BOOLEAN-EXP    : map(while-loop, boolean-expression)     = {}
var START-RANGE        : map(for-loop, expression)                = {}
var END-RANGE          : map(for-loop, expression)                 = {}

```

```

form LOOP-STATEMENT-TREE-ATTRIBUTES
  define-tree-attributes('basic-loop, {'loop-id,
                                'loop-statement-sequence});

  define-tree-attributes('for-loop, {'loop-id,
                                'aname-use,
                                'start-range,
                                'end-range,
                                'loop-statement-sequence});

  define-tree-attributes('while-loop, {'loop-id,
                                'loop-boolean-exp,
                                'loop-statement-sequence})

```

```

%%% -----
%%% Exit-Statement
%%%-----
var EXIT-ID           : map(Exit-statement, identifier-definition) = {}

```

```

form EXIT-STATEMENT-TREE-ATTRIBUTES
  define-tree-attributes('exit-statement, {'exit-id, 'expression-value})

```

```

%%% -----
%%% Read and Write-Statment
%%%-----
var READ-VALUE        : map(Read-statement, variable-use)         = {}
var WRITE-EXPRESSION   : map(Write-statement, expression)         = {}

```

```

form IO-TREE-ATTRIBUTES
  define-tree-attributes('Read-statement, {'read-value});
  define-tree-attributes('Write-statement, {'write-expression})

```

```

%%% -----
%%% Return-Statment
%%%-----

```

```

form RETURN-STATEMENT-TREE-ATTRIBUTES
  define-tree-attributes('return-statement, {'expression-value})

```

```

%%% -----
%%% Block-Structure
%%%-----

```

```

form BLOCK-STRUCTURE-TREE-ATTRIBUTES
  define-tree-attributes('Block-structure, {'adefined-name,

```

```
'declaration-part,  
'statement-sequence,  
'adefined-name}))
```

```
%%%-----  
%%% Procedure-call  
%%% -----
```

```
form PROCEDURE-CALL-TREE-ATTRIBUTES  
  define-tree-attributes('Procedure-call, {'aname-use}))
```

## B.2 OML with Ada PDL Grammar

### B.2.1 OML Grammar

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% File-Name : oml-gm.re   (OML grammar productions)
%%%
%%% Authors : Capt Mary Boom, Capt Brad Mallare
%%%
%%% Purpose : This file builds the productions that define the grammar
%%%           for an OML specification. When parsing OML specifications, these
%%%           productions ensure that the specifications satisfy the syntax
%%%           requirements defined in the OML BNF.
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

!! in-package("ru")
!! in-grammar('syntax')

grammar OML

  inherits-from AdaCs

  start-classes informal-model

  file-classes informal-model

  productions

    informal-model ::=
      ["specification" defined-name analysis-obj-map + "" ]
      builds informal-model,

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ENTITY OBJECT PRODUCTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    entity-class ::=
      [defined-name "class-of" "entity"
       "type" ":" ([ "external" !! external-entity ] | "internal")
       {["parts" entity-user-decl-map + ";"]}
       {["constraints" constraints-map + ";" ]}]
      builds entity-class,

    entity-instance ::=
      [defined-name "instance-of"
       ( ["entity"
        "type" ":" ([ "external" !! external-entity ] | "internal")
        {["parts" entity-user-decl-map + ";"]}
        | [name-use
         {["values" entity-user-def-map + ";"]}]]
       )
       {["constraints" constraints-map + ";"]}
      builds entity-instance,

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% USER-DECLARED-ATTRIBUTE-PRODUCTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    user-declared-attr ::=
```

```

[defined-name ":" variable-type
{"range" "{" range-attribute "}"}]
{"init-val" (value-map | symbol-val)}] ]
builds user-declared-attr,

```

```

    user-defined-attr ::=
[name-use ":" (value-map | symbol-val)]
builds user-defined-attr,

```

```

    enumerated-range ::=
[enum-range-map + "," ]
builds enumerated-range,

```

```

    integer-range ::=
[integer-low-map "." "." integer-high-map]
builds integer-range,

```

```

    real-range ::=
[real-low-map "." "." real-high-map]
builds real-range,

```

XXXXXXXXXXXXXXXXXXXXXXXXX PROCESS OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

    process-object ::=
[defined-name "instance-of" "process"
{"constraints" constraints-map + ";"}]
builds process-object,

```

XXXXXXXXXXXXXXXXXXXXXXXXX STATE OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

    state-object ::=
[defined-name "instance-of" "state"
"state-space" ":" state-space-map + ";"
{"constraints" constraints-map + ";"}]
builds state-object,

```

XXXXXXXXXXXXXXXXXXXXXXXXX BEHAVIOR OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

    decision-table ::=
[defined-name "instance-of" "behavior"
[dt-cond-row-map + ";" ] "-->"
{"dt-action-row-map + ";" }]
["event" "," dt-event-map + "," ]
{"constraints" constraints-map + ";" }]
builds decision-table,

```

```

    condition-row ::=
[name-use "," condition-entry-map + "," ]
builds condition-row,

```

```

    action-row ::=
[name-use "," action-entry-map + "," ]
builds action-row,

```

% Name use in the next production allows us to have symbols such as 'safe and  
% 'unsafe in the decision tables. It is not a pure use of the map name-use.

```

condition-entry ::=
  [ (["dont-care" !! dont-care-value] | condition-range | name-use) ]
  builds condition-entry,

action-entry ::=
  [ ( action-value | name-use | action-expr) ]
  builds action-entry,

pre-post-cond ::=
  [defined-name "instance-of" "behavior"
    behavior-ppc-map + ";"
    {["constraints" constraints-map + ";"]} ]
  builds pre-post-cond,

ppc-statement ::=
  [ppc-pre-map + "&" "-->" { [ppc-post-map + "&"] }
    "event" ppc-event-map ]
  builds ppc-statement,

process-desc-lang ::=
  [defined-name "instance-of" "behavior" pdl-map
    {["constraints" constraints-map + ";"]} ]
  builds process-desc-lang,

```

**XXXXXXXXXXXXXXXXXXXXX STORE OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX**

```

store-object ::=
  [defined-name "instance-of" "store"
    "nature" ":" (["set" !! nature-set-map] | "sequence")
    "content" ":" content-name
    {["key" ":" key-name]}
    {["order" ":" (["ascending" !! order-set-map] | "descending")]}
    {["constraints" constraints-map + ";"]} ]
  builds store-object,

```

**XXXXXXXXXXXXXXXXXXXXX RELATIONSHIP OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX**

```

relationship-object ::=
  [defined-name "instance-of" "relationship"
    "type" ":" rel-type-map
    "cardinality" ":" cardinality-map
    {["constraints" constraints-map + ";"]} ]
  builds relationship-object,

```

**XXXXXXXXXXXXXXXXXXXXX FLOW OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX**

```

flow-object ::=
  [defined-name "instance-of" "flow"
    "flow-link" ":" flow-link-map
    "flow-data" ":" flow-data-map
    {["constraints" constraints-map + ";"]} ]
  builds flow-object,

```

**XXXXXXXXXXXXXXXXXXXXX EVENT OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX**

```

event-object ::=
  [defined-name "instance-of" "event"
   "type" ":" ([ "internal" !! event-type ] | "external")
   {"priority" ":" event-priority}]
   {"constraints" constraints-map + ";" }]
builds event-object,

```

XXXXXXXXXXXXXXXXXXXXX RELATION-TABLE OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX

```

relation-table ::=
  [defined-name "instance-of" "relation-table"
   relation-sequence + ";"
   {"constraints" constraints-map + ";" }]
builds relation-table,

relation ::=
  [from-obj-map "," assoc-obj-map "," to-obj-map]
builds relation,

```

XXXXXXXXXXXXXXXXXXXXX IDENTIFIER OBJECT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX

```

identifier-def ::= [ name ]
builds identifier-def,

identifier-use ::= [ name ]
builds identifier-use,

```

XXXXXXXXXXXXXXXXXXXXX LITERAL CONSTANT PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX

```

integer-literal ::= [ integer-value-of ]
builds integer-literal,

real-literal ::= [ real-value-of ]
builds real-literal,

true-literal ::= [ "true" ]
builds true-literal,

false-literal ::= [ "false" ]
builds false-literal,

string-literal ::= [ string-value-of ]
builds string-literal,

```

XXXXXXXXXXXXXXXXXXXXX VARIABLE TYPE PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXX

```

type-boolean ::= [ "boolean" ]
builds type-boolean,

type-string ::= [ "string" ]
builds type-string,

type-symbol ::= [ "symbol" ]
builds type-symbol,

```

```

type-integer ::= [ "integer" ]
               builds type-integer,

type-real    ::= [ "real" ]
               builds type-real,

type-set     ::= [ "set" ]
               builds type-set,

type-sequence ::= [ "sequence" ]
                  builds type-sequence,

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX FLOW TYPE PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

flow-pp ::= [ "proc-proc" ]
           builds flow-pp,

flow-ps ::= [ "proc-store" ]
           builds flow-ps,

flow-sp ::= [ "store-proc" ]
           builds flow-sp,

flow-pe ::= [ "proc-entity" ]
           builds flow-pe,

flow-ep ::= [ "entity-proc" ]
           builds flow-ep,

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX CARDINALITY TYPE PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

One-Many ::= [ "1-m" ]
            builds One-Many,

One-One  ::= [ "1-1" ]
            builds One-One,

Many-One ::= [ "m-1" ]
            builds Many-One,

Many-Many ::= [ "m-m" ]
            builds Many-Many,

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX EXPRESSION PRODUCTIONS XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

XXX Arithmetic Expressions XXX

```

arithmetic-add ::= [ argument-1 "+" argument-2 ] builds arithmetic-add,

unary-minus   ::= [ "-" argument-1 ] builds unary-minus,

arithmetic-divide ::= [ argument-1 "/" argument-2 ]
                    builds arithmetic-divide,

arithmetic-multiply ::= [ argument-1 "*" argument-2 ]
                      builds arithmetic-multiply,

```

```

arithmetic-subtract ::= [ argument-1 "-" argument-2 ]
    builds arithmetic-subtract,

```

### %%% Boolean Expressions %%%

```

boolean-and ::= [ argument-1 "and" argument-2 ]
    builds boolean-and,

```

```

boolean-not ::= [ "not" argument-1 ]
    builds boolean-not,

```

```

boolean-or ::= [ argument-1 "or" argument-2 ]
    builds boolean-or,

```

```

compare-equal ::= [ argument-1 "=" argument-2 ]
    builds compare-equal,

```

```

compare-greater-or-equal ::= [ argument-1 ">=" argument-2 ]
    builds compare-greater-or-equal,

```

```

compare-greater-than ::= [ argument-1 ">" argument-2 ]
    builds compare-greater-than,

```

```

compare-less-or-equal ::= [ argument-1 "<=" argument-2 ]
    builds compare-less-or-equal,

```

```

compare-less-than ::= [ argument-1 "<" argument-2 ]
    builds compare-less-than,

```

```

compare-not-equal ::= [ argument-1 "/=" argument-2 ]
    builds compare-not-equal,

```

```

compare-in ::= [ argument-1 "in" argument-2 ] %% arg-2 must be a set,
    builds compare-in,                    %% seq or variable

```

```

compare-for-all ::= ["forall" "(" name-uses + "," ")"
    "(" [set-arg + "&"] "=>" argument-1 ")"]
    builds compare-for-all,

```

```

compare-exists ::= ["exists" "(" name-uses + "," ")"
    "(" [set-arg + "&"] ")"]
    builds compare-exists,

```

### %%% Set comprehension expressions %%%

```

set-union ::= [argument-1 "union" argument-2]
    builds set-union,

```

```

set-diff ::= [argument-1 "set-diff" setbuilder-map ]
    builds set-diff,

```

```

getitem ::= [ "getitem" "(" setbuilder-map ")" ]
    builds getitem,

```

```

getset ::= [ "getset" setbuilder-map ]

```



```

        builds getset,

setbuilder ::= [ "{" defined-name "|" set-diff-condition + "&" "]" ]
        builds setbuilder,

%%% Predicate Range Expressions %%%

r-equal ::= ["=" argument ]
        builds r-equal,

r-greater-or-equal ::= [ ">=" argument ]
        builds r-greater-or-equal,

r-greater-than ::= [ ">" argument ]
        builds r-greater-than,

r-less-or-equal ::= [ "<=" argument ]
        builds r-less-or-equal,

r-less-than ::= [ "<" argument ]
        builds r-less-than,

r-not-equal ::= [ "/=" argument ]
        builds r-not-equal,

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
assignment-statement ::= [ LHS ":" RHS ]
        builds assignment-statement,

create ::= ["create" "(" defined-name ":" name-use ")"]
        builds create,

destroy ::= ["destroy" "(" name-use ")"]
        builds destroy,

display ::= ["display" "(" (name-use | display-set ) ")"] ]
        builds display

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
no-patterns

precedence

    for expression brackets "(" matching ")"

    (same-level "and", "or" associativity left),
    (same-level "<", "<=", "=", ">=", ">", "/=" associativity none),
    (same-level "in", "set-diff", "union" associativity left),
    (same-level "+", "-" associativity left),
    (same-level "*", "/", "mod" associativity left),
    (same-level "**", "abs", "not" associativity none)

symbol-start-chars
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

```

```
symbol-continue-chars
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-."

comments
  "%" matching "
",
  "##|" matching "||#" nested

end
```

### B.2.2 Ada PDL Grammar

```
!! in-package("ru")
!! in-grammar('syntax')
```

```
XXX -*- Mode: RE; Package: ADA; Base: 10; Syntax: Refine -*-
```

```
#||
```

This file defines the grammar for parsing a subset of Ada called AdaCs.

```
||#
```

grammar AdaCs

start-classes procedure-declaration

file-classes procedure-declaration

productions

arithmetic-add ::= [ argument-1 "+" argument-2 ] builds arithmetic-add,

unary-plus ::= [ "+" argument-1 ] builds unary-plus,

unary-minus ::= [ "-" argument-1 ] builds unary-minus,

arithmetic-divide ::= [ argument-1 "/" argument-2 ]  
builds arithmetic-divide,

arithmetic-modulo ::= [ argument-1 "mod" argument-2 ]  
builds arithmetic-modulo,

arithmetic-multiply ::= [ argument-1 "\*" argument-2 ]  
builds arithmetic-multiply,

arithmetic-subtract ::= [ argument-1 "-" argument-2 ]  
builds arithmetic-subtract,

arithmetic-abs ::= [ "abs" argument-1 ]  
builds arithmetic-abs,

arithmetic-exponent ::= [ argument-1 "\*\*" argument-2 ]  
builds arithmetic-exponent,

aassignment-statement ::= [ ALHS "==" expression-value ";" ]  
builds aassignment-statement,

boolean-and ::= [ argument-1 "and" argument-2 ]  
builds boolean-and,

boolean-not ::= [ "not" argument-1 ]

```

        builds boolean-not,

boolean-or ::= [ argument-1 "or" argument-2 ]
        builds boolean-or,

compare-equal ::= [ argument-1 "=" argument-2 ]
        builds compare-equal,

compare-greater-or-equal ::= [ argument-1 ">=" argument-2 ]
        builds compare-greater-or-equal,

compare-greater-than ::= [ argument-1 ">" argument-2 ]
        builds compare-greater-than,

compare-less-or-equal ::= [ argument-1 "<=" argument-2 ]
        builds compare-less-or-equal,

compare-less-than ::= [ argument-1 "<" argument-2 ]
        builds compare-less-than,

compare-not-equal ::= [ argument-1 "/" argument-2 ]
        builds compare-not-equal,

afunction-call ::= [ aname-use "(" ]
        builds afunction-call,

procedure-call ::= [ aname-use "(" ";" ]
        builds procedure-call,

identifier-definition ::= [ name ]
        builds identifier-definition,

identifier-use ::= [ name ]
        builds identifier-use,

integer-literal ::= [ integer-value-of ]
        builds integer-literal,

real-literal ::= [ real-value-of ]
        builds real-literal,

true-literal ::= [ "true" ]
        builds true-literal,

false-literal ::= [ "false" ]
        builds false-literal,

an-if-statement ::= [ "if" test-condition "then" then-part + ""
        {([elseif-part + "" "else" else-part + "" ]
        | ["else" else-part + ""]))"end" "if" ";" ]
        builds an-if-statement,

elseif-statement
        ::= [ "elseif" elseif-test-condition "then" elseif-statements + "" ]
        builds elseif-statement,

```

```

basic-loop
  ::= [ {[loop-id ":" ]} "loop" loop-statement-sequence + ""
    "end" "loop" ";"]
    builds basic-loop,

for-loop
  ::= [ {[loop-id ":" ]} "for" aname-use "in" start-range "." "."
    end-range "loop" loop-statement-sequence + "" "end" "loop" ";"]
    builds for-loop,

while-loop
  ::= [ {[loop-id ":" ]} "while" loop-boolean-exp "loop"
    loop-statement-sequence + "" "end" "loop" ";"]
    builds while-loop,

return-statement ::= [ "return" expression-value ";"]
    builds return-statement,

exit-statement ::= [ "exit" {exit-id} {"when" expression-value} ";"]
    builds exit-statement,

procedure-declaration ::= ["procedure" adefined-name "is" block-map]
    builds procedure-declaration,

function-declaration
  ::= ["function" adefined-name "return" avariable-type "is"
    block-map]
    builds function-declaration,

block-structure
  ::= [{[adefined-name ":" ]} {"declare" declaration-part * ""}]
    "begin" statement-sequence + "" "end" {adefined-name} ";"]
    builds Block-structure,

ablock
  ::= [declaration-part * "" "begin" statement-sequence + "" "end"
    {bdefined-name} ";"]
    builds ablock,

variable-declaration
  ::= [adefined-names + "," ":" {constant-option} avariable-type
    {"!=" expression-value} ";"]
    builds variable-declaration,

enumerated-declaration
  ::= ["type" adefined-name "is" "(" adefined-names + "," ")" ";"]
    builds enumerated-declaration,

succ-expression ::= [enumtype-name "succ" "(" aname-use ")"]
    builds succ-expression,

pred-expression ::= [enumtype-name "pred" "(" aname-use ")"]
    builds pred-expression,

val-expression ::= [enumtype-name "val" "(" expression-value ")"]
    builds val-expression,

```

```

char-expression ::= [aname-use "'char"]
    builds char-expression,

float-to-int ::= ["integer" "(" expression-value ")"]
    builds float-to-int,

int-to-float ::= ["float" "(" expression-value ")"]
    builds int-to-float,

subprogram-declaration ::= [( subprog2proc-decl | subprog2func-decl )]
    builds subprogram-declaration,

type-enumerated ::= [ enumerated-symbol-map ]
    builds type-enumerated,

type-boolean ::= [ "boolean" ]
    builds type-boolean,

type-string ::= [ "string" ]
    builds type-string,

type-integer ::= [ "integer" ]
    builds type-integer,

type-float ::= [ "float" ]
    builds type-float,

Constant-flag ::= ["constant"]
    builds Constant-flag,

Read-statement ::= ["read" "(" read-value ")" ";"]
    builds Read-statement,

Write-statement
    ::= ["write" "(" write-expression ")" ";"]
    builds Write-statement

no-patterns

precedence

    for expression brackets "(" matching ")"

    (same-level "and", "or" associativity left),
    (same-level "<", "<=", "=", ">=", ">", "/" associativity none),
    (same-level "+", "-" associativity left),
    (same-level "*", "/", "mod" associativity left),
    (same-level "**", "abs", "not" associativity none)

symbol-start-chars
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_"

symbol-continue-chars
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_"

```

```
comments
  "--" matching "
"  %% "--" to end-of-line

%% brackets "(" matching ")",
%%      "[" matching "]"

end
```

## Appendix C. Object Modeling Language REFINE Implementation

### C.1 Translation Software

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%%%                                                                 %%%
%%% File-Name : Trans-Oml.re                                         %%%
%%%                                                                 %%%
%%% Authors : Capt Mary Boom, Capt Brad Mallare                     %%%
%%%                                                                 %%%
%%% Purpose : This program parses an OML specification into a Refine %%%
%%%             Abstract Syntax Tree (AST) and then translates the AST %%%
%%%             representation into an executable Refine source code program. %%%
%%%                                                                 %%%
%%% Dependencies : Prior to compiling this program, the following programs %%%
%%%             must be compiled and loaded into the Refine database: %%%
%%%                                                                 %%%
%%%             Dialect                                             %%%
%%%             dm.re                                               %%%
%%%             gm.re                                               %%%
%%%             r-lib.re                                           %%%
%%%             lisp-utilities.lisp                                %%%
%%%                                                                 %%%
%%% Functions :                                                     %%%
%%%             Convert                                             %%%
%%%             Trans                                              %%%
%%%             Print-Class-Decl                                    %%%
%%%             Print-Instance-Decl                                %%%
%%%             Print-Attr-Value                                    %%%
%%%             Print-Store-Function                               %%%
%%%             Print-Flow-Function                                %%%
%%%             Print-Behavior-Rule                                 %%%
%%%             Print-Decision-Table                               %%%
%%%             Print-Pre-Post-Condition                           %%%
%%%             Convert-Attr-Name                                   %%%
%%%             Find-State                                          %%%
%%%             Print-Action-Entry                                  %%%
%%%             Print-State-Function                               %%%
%%%             Print-State-Behaviors                              %%%
%%%             Print-Proc-Function                                %%%
%%%             Add-State-Sim-Function                             %%%
%%%             Add-Proc-Sim-Function                              %%%
%%%             Print-Predicate-Expression                         %%%
%%%             Print-Statement                                     %%%
%%%             Print-Expression                                    %%%
%%%             Print-Delete-Object-Function                       %%%
%%%                                                                 %%%
%%% Operation : After loading this program and the required programs %%%
%%%             mentioned above, type the following commands at the Refine prompt: %%%
%%%                                                                 %%%
%%%             (convert "<your-OML-file-name>")                    %%%
%%%                                                                 %%%
%%% The name of the generated executable specification will be displayed %%%
%%% on the screen. Additionally, the executable specification will be %%%
%%% automatically compiled and loaded.                             %%%
%%%                                                                 %%%

```



```

%%% Once the OML specification has been converted into an executable %%%
%%% Refine specification, the specification can be executed by typing %%%
%%% the following command: %%%
%%%                                     (sim) %%%
%%%                                     %%%
%%% If a control structure was not provided in the OML spec, then the %%%
%%% user will be prompted to make control decisions during the simulation %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!! in-package ('RU)
!! in-grammar ('user)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% GLOBAL VARIABLES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Global variables are in all CAPS throughout this file

type ATTR-REF = tuple(old-name : string, uniq-name : string)
var OML-OBJ : object-class subtype-of user-object
var ATTR-NAME-TABLE : set(ATTR-REF) = {}      %% attribute name table
var ROOT : object = undefined                %% root of the AST
var EXT-EV-SET : set(symbol) = {}            %% set of all external events
var INITIAL-STATE : symbol = undefined       %% start state=1st state in OML file
var TILDE : string = ""
var OBJ-NAMES : set(symbol) = {}             %% set of all object instance names
var PROC-NAMES : set(symbol) = {}            %% set of all process object names
var FLOW-NAMES : set(tuple(name : symbol, flow-type : symbol)) = {}
                                                    %% set of all flow names and types
var STORE-NAMES : set(symbol) = {}
var EXTERNALS-NAMES : set(symbol) = {}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CONVERT %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% This is the main function. It parses in the OML file and calls the
%%% translation function (trans). The converted file is then compiled and
%%% loaded into the Refine object base.

function convert(filename : string) =

  (let(source-file : string = ""))

  %% parses in OML spec and ensures that the top object is an informal model
  if informal-model(pf(filename)) then

    (ROOT <- oml-root());                      %% find the root of the AST

    %% (if sem(ROOT) then                      %% semantic checks the file
    %% (if tcheck(ROOT) then                  %% type checks the file
    %%   source-file <- trans(ROOT);
    %%   compile-file(source-file);
    %%   load(source-file);
    %%   format(true,"To run, keyin \"(sim)\" \"%\"") )
    %% else
    %%   format(true,"You must fix type mismatches before compilation"))
    %% else
    %%   format(true,"You must fix semantic errors before compilation"))))
    %%   source-file <- trans(ROOT) )

```

```

    else format(true, "file must begin with a specification declaration")
  )

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%%% This function enumerates over the various objects in the AST and calls the
%%% appropriate functions which translate the AST objects into a REFINES
%%% executable.

function trans(o:object) =

  let(Fname : string = "oml",
      New-Fname : string = "oml.re")

  (if informal-model(o)
   then
     Fname <- princ-to-string(name(defined-name(o)));
     New-Fname <- concat(Fname, ".re");
     format(true, "Translating to ~A ....~%", New-Fname);

     (enumerate obj over
      [obj | (obj : OML-OBJ) OML-OBJ(obj)] do
        erase-object(obj) );
      PROC-NAMES <- {});

    rd-on(New-Fname); %redirect standard output to file

    format(true, "!! in-package ('RU)~%");
    format(true, "!! in-grammar ('user)~%~%");

    format(true, "var OML-Obj : object-class subtype-of user-object~%~%");
    format(true, "var ~A : object-class subtype-of OML-Obj~%~%", Fname);
    (if ex (x) (State-Object(x)) then
      format(true, "type return-values = tuple(validity: symbol,
        events: seq(symbol),
        behaviors : seq(symbol),
        st-behaviors : seq(symbol))~%~%")
    elseif ex (x) (Process-Object(x)) then
      format(true, "type return-values = tuple(validity: symbol,
        next-procs : seq(symbol))~%~%")
    );

    format (true, "\\%\\%\\% Define object classes ~%~%");

    (enumerate vars over
     descendants-of-class(o, 'Entity-Class) do
       print-class-decl(vars, Fname);
       if defined?(external-entity(vars)) then
         EXTERNALS-NAMES <-
           EXTERNALS-NAMES with name(defined-name(vars)) );

    format (true, "\\%\\%\\% Define instances of object classes ~%~%");

    (enumerate instance over
     descendants-of-class(o, 'Entity-Instance) do
       print-instance-decl(instance, Fname);

```

```

      OBJ-NAMES <- OBJ-NAMES with name(defined-name(instance));
      if defined?(external-entity(instance)) then
        EXTERNALS-NAMES <-
          EXTERNALS-NAMES with name(defined-name(instance));

      format (true, "\\%\\%\\% Define Store Objects ~%~%");

      (enumerate sto over
descendants-of-class(o,'Store-Object) do
      print-store-function(sto);
        STORE-NAMES <- STORE-NAMES with name(defined-name(sto)));

      format (true, "\\%\\%\\% Define objects for each flow object ~%~%");

      (enumerate flow over
descendants-of-class(o,'Flow-Object) do
      print-flow-function(flow);
        OBJ-NAMES <- OBJ-NAMES with name(defined-name(flow));
        FLOW-NAMES <- FLOW-NAMES
          with (<name(defined-name(flow)),name(flow-data-map(flow))>));

      format (true, "\\%\\%\\% Define functions for behavior objects ~%~%");

      (enumerate beh over
descendants-of-class(o,'Behavior-Object) do
      print-behavior-rule(beh));

%% create a set of all external events
      (enumerate ev over
descendants-of-class(o,'Event-Object) do
        if undefined?(event-type(ev)) then      %% checks if external event
          EXT-EV-SET <- EXT-EV-SET with name(defined-name(ev)) );

      format (true, "\\%\\%\\% Define function for each state object ~%~%");

      (enumerate state over
descendants-of-class(o,'State-Object) do
      print-state-function(state);
        if state = last(set-to-seq(descendants-of-class(o,'State-Object)))
          then INITIAL-STATE <- name(defined-name(state)) );

      (if ~empty(descendants-of-class(o,'State-Object)) then
        Add-State-Sim-Function(Fname));

      format (true, "\\%\\%\\% Define function for each process object ~%~%");

      (enumerate proc over
descendants-of-class(o,'Process-Object) do
        PROC-NAMES <- PROC-NAMES with name(defined-name(proc));
        print-proc-function(proc));

      (if ~empty(descendants-of-class(o,'Process-Object)) then
        Add-Proc-Sim-Func());

      format (true, "

```

```

\\%\\%\\% Defines function for erasing all objects in Refine's database.
\\%\\%\\% Execute this function before you reload this file if you do not use
\\%\\%\\% the convert process. ~%~%");

Print-Delete-Object-Function(Fname);

rd-off();      %% return to writing standard output

values()      %%prevents "nil" from being returned at end of program

else format(true, "file must begin with a specification declaration"));
princ-to-string(name(defined-name(o))) % return file name sans ".re"

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%% Top-obj is the top level object name in the specification
%% Class-name is the current object-class name. If the object class is a
%% class-of an entity object, then the object class name will be used to
%% identify it. If the object class is an instance-of an entity object then
%% the class name concatenated with "-ENTITY" will be used to identify that
%% object class.

function print-class-decl (vd : object, top-obj : string) =

%%% This section prints out the object class definitions

let(class-name : string = undefined,
    attr-name : string = undefined,
    u-attr-name : string = undefined,
    x-ref : ATTR-REF = <undefined, undefined>)

class-name <- princ-to-string(name(defined-name(vd)));

(if entity-class(vd) then      %% it is a class of an entity object
    format (true, "var ~A : ", class-name);
    format (true, "object-class subtype-of ~A ~%", top-obj)

else                            %% it is an instance of an entity object
    class-name <- concat(princ-to-string(name(defined-name(vd))), "-ENTITY");
    format (true, "var ~A : ", class-name);
    format (true, "object-class subtype-of ~A ~%", top-obj)
);

% This section declares the attributes of each object class. Each attribute
% is declared as a variable which maps the object class to the attribute type.
% The attribute names are translated into the class-name concatenated with the
% attribute name to ensure that all variable names are unique.

(if defined?(entity-user-decl-map(vd)) then
    (enumerate id over entity-user-decl-map(vd) do
        attr-name <- princ-to-string(name(defined-name(id)));
        u-attr-name <- concat(class-name, "-", attr-name);
        format (true, "var ~A: ", u-attr-name);
        format (true, " map(~A, ", class-name);
        ( if type-integer(variable-type(id)) then

```

```

        format(true, "integer")
    elseif type-boolean(variable-type(id)) then
        format(true, "boolean")
    elseif type-real(variable-type(id)) then
        format(true, "real")
    elseif type-string(variable-type(id)) then
        format(true, "string")
    elseif type-symbol(variable-type(id)) then
        format(true, "symbol")
    else format(true, "type declaration error")
);
format(true, ") = {||}~%" );

x-ref.old-name <- concat(class-name, ".", attr-name);
x-ref.uniq-name <- concat(u-attr-name, "(", class-name, ")");
ATTR-NAME-TABLE <- union(ATTR-NAME-TABLE, {x-ref})
);
format (true, "~%"
)

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%% This function creates instances of class objects. Instances are made by
%% creating a FORM statement which defines the instance object and sets the
%% attributes for that object.

function print-instance-decl (inst : object, file-name : string) =

    let(class-name : string = undefined,
        object-name : string = undefined,
        attr-name : string = undefined,
        u-attr-name : string = undefined,
        x-ref : ATTR-REF = <undefined, undefined>)

    %%% If an object is an instance (as opposed to a class) of Entity, then we must
    %%% declare an object class as well as instantiate that class. This section
    %%% declares object classes for instances of entity objects

    (if undefined?(name-use(inst)) then
        print-class-decl(inst, file-name)
    );

    %%% This section creates an instance of the object class

    object-name <- princ-to-string(name(defined-name(inst)));

    (if defined?(name(name-use(inst))) then %% it is an instance of a class
        class-name <- princ-to-string(name(name-use(inst)));
        format (true, "var ~A : ~A = ~%",
            name(defined-name(inst)), name(name-use(inst)));
        format (true, " set-attrs(make-object('~A'),~%", name(name-use(inst)));
        (if defined?(entity-user-def-map(inst)) then
            format (true, " 'name, '~A,~%", name(defined-name(inst)))
        else
            format (true, " 'name, '~A~%", name(defined-name(inst)))
        )
    )

```

```

else
    %% it is an instance of an entity
    class-name <- princ-to-string(name(defined-name(inst)));
    format (true, "var ~A : ~A = ~%",
            name(defined-name(inst)), concat(class-name, "-ENTITY"));
    format (true, "  set-attrs(make-object('~A'),~%"
            ,concat(class-name, "-ENTITY"));
    (if defined?(entity-user-decl-map(inst)) then
        format (true, "      'name, '~A,~%", name(defined-name(inst)))
    else
        format (true, "      'name, '~A~%", name(defined-name(inst)))
    )
);
(if defined?(entity-user-def-map(inst)) then
    enumerate attr over entity-user-def-map(inst) do
        attr-name <- princ-to-string(name(name-use(attr)));
        u-attr-name <- concat(class-name, "-", attr-name);
        format (true, "      '~A, ", u-attr-name);
        print-attr-value(attr);

%% this creates a table that cross-references attribute names used in behaviors
%% with the names as they are actually stored in the system

        x-ref.old-name <- concat(object-name, ".", attr-name);
        x-ref.uniq-name <- concat(u-attr-name, "(", object-name, ")");
        ATTR-NAME-TABLE <- union(ATTR-NAME-TABLE, {x-ref});

        (if attr neq last(entity-user-def-map(inst)) then
            format(true, ",~%")
        else
            format (true, ")~%")
        )

elseif defined?(entity-user-decl-map(inst)) then
    enumerate attr over entity-user-decl-map(inst) do
        attr-name <- princ-to-string(name(defined-name(attr)));
        u-attr-name <- concat(class-name, "-ENTITY-", attr-name);
        format (true, "      '~A, ", u-attr-name);
        print-attr-value(attr);

%% this creates a table that cross-references attribute names used in behaviors
%% with the names as they are actually stored in the system

        x-ref.old-name <- concat(class-name, ".", attr-name);
        x-ref.uniq-name <- concat(u-attr-name, "(", class-name, ")");
        ATTR-NAME-TABLE <- union(ATTR-NAME-TABLE, {x-ref});

        (if attr neq last(entity-user-decl-map(inst)) then
            format(true, ",~%")
        else
            format (true, ")~%")
        )
    );
    format(true, " ~%")

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
PRINT-ATTR-VALUE XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```



```

        name(defined-name(flow-obj)), name(flow-data-map(flow-obj)));
format (true, "  set-attrs(make-object('~A'),~%"
                                ,name(flow-data-map(flow-obj)));
format (true, "          'name, '~A)~%"~%", name(defined-name(flow-obj)));

class-obj <- arb({ x | (x) x in analysis-obj-map(root) & entity-class(x) &
                    name(defined-name(x)) = name(flow-data-map(flow-obj))});

(if defined?(entity-user-decl-map(class-obj)) then
  enumerate attr over entity-user-decl-map(class-obj) do
    attr-name <- princ-to-string(name(defined-name(attr)));
    class-name <- princ-to-string(name(flow-data-map(flow-obj)));
    u-attr-name <- concat(class-name, "-", attr-name);
    x-ref.old-name <- concat(obj-name, ".", attr-name);
    x-ref.uniq-name <- concat(u-attr-name, "(", obj-name, ")");
    ATTR-NAME-TABLE <- union(ATTR-NAME-TABLE, {x-ref}) )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PRINT-BEHAVIOR-RULE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Determines the type of behavior object (i.e. PDL, DT, or PPC) and
%% translates the behavior into a Refine function with transform rules
%% Print flag denotes whether a NON dont-care value has been found and printed

function print-behavior-rule (o : object) =

%%% Converts a DT into a Refine function %%%

  if decision-table(o) then
    print-decision-table(o);
    format(true, "%\%\%\%\%\%\%\%\%\%\%\%\%\%\% ~%"~%")

%% Converts a PPC into a Refine function

  elseif pre-post-cond(o) then
    print-pre-post-condition(o)

%% Converts a PDL into a Refine function

% elseif process-desc-lang(o) then
%   trans-ada(o)          %% calls PDL translation function in trans-ada.re

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PRINT-DECISION-TABLE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% Translates DT objects into Refine

function print-decision-table(o : object) =

  let(num-rules : integer = undefined,
      print-flag : boolean = false)

    num-rules <- size(condition-entry-map(dt-cond-row-map(o)(1)));

%% prints out the function name and declares a local variable
format(true, "function ~A() : symbol =~%", name(defined-name(o)));
format(true, "  let(return-symbol : symbol = undefined)~%"~%");

```



```

format (true, "~%");

( enumerate i from 1 to num-rules do
  print-flag <- false;                                %% reset print-flag

%% prints out preconditions for each rule
(enumerate row over dt-cond-row-map(o) do
  %% if entry is not dont-care
  if undefined?(dont-care-value(condition-entry-map(row)(i))) then
    (if print-flag then
      format (true, " & ");
      format (true, "~A ", convert-attr-name(name(name-use(row))));
      print-predicate-expression(condition-range(
        condition-entry-map(row)(i)));
      format(true, "~%")
    else
      format (true, " (~A ", convert-attr-name(name(name-use(row))));
      print-predicate-expression(condition-range(
        condition-entry-map(row)(i)));
      format(true, "~%");
      print-flag <- true
    )
  );
format (true, " -->~%");                                %% print out the transform symbol

%% print out the postconditions for each rule
print-flag <- false;                                %% reset print-flag
(if defined?(dt-action-row-map(o)) then
  (enumerate row over dt-action-row-map(o) do
    if print-flag then
      format (true, " & ");
      format (true, "~A ", convert-attr-name(name(name-use(row))));
      format (true, "<- ");
      print-action-entry(action-entry-map(row)(i));
      format(true, "~%")
    else
      format (true, " (~A ", convert-attr-name(name(name-use(row))));
      format (true, "<- ");
      print-action-entry(action-entry-map(row)(i));
      format(true, "~%");
      print-flag <- true
    )
  );

%% returns the name of the next state if a next event is specified

(if name(dt-event-map(o)(i)) ~= 'none then
  let (next-state : symbol = undefined)
  next-state <-
    find-state(name(defined-name(o)), name(dt-event-map(o)(i)));
  (if print-flag then
    format (true, " & ");
    format (true, "(return-symbol <- '~A", next-state)
  else

```

```

        format (true, " (return-symbol <- '~A", next-state)
    ));

%% add semicolon between transforms (rules) and line feeds
    (if i ~= num-rules then
        format (true, ");~%~%")
    else
        format (true, ")~%~%")
    );
    format (true, ");~%return-symbol~%")

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PRINT-PRE-POST-CONDITION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% Translates PPC objects into Refine. Currently converted into If-Then
%%% structure. Once PDL is fully integrated, recommend PPCs be converted into
%%% transform construct which is a better description of "what" must be done.

function print-pre-post-condition(o : object) =

    format(true, "function ~A() =~%", name(defined-name(o)));
    format(true, " let(return-symbol : symbol = undefined)~%~%");
    format (true, " ( ");
    (enumerate ppc over behavior-ppc-map(o) do

        %% prints out preconditions for each rule
        (enumerate precondition over ppc-pre-map(ppc) do
            (if precondition = first(ppc-pre-map(ppc)) then
                format (true, "(if ");
                print-expression(precondition);
                format(true, "~%")
            else
                format (true, " and ");
                print-expression(precondition);
                format(true, "~%")
            )
        ); %% ends enum precondition

        format (true, " then~%");          %% print out the transform symbol

    %% prints out postconditions for each rule
    (if defined?(ppc-post-map(ppc)) then
        ( enumerate postcond over ppc-post-map(ppc) do
            (if postcond ~= last(ppc-post-map(ppc)) then
                format(true, " (");
                print-statement(postcond);
                format(true, ");~%")
            else
                format(true, "(");
                print-statement(postcond);
                format(true, ")~%")
            )
        ) %% ends enum postcond
    );

    %% if a next event is specified, find the name of the next state associated
    %% with that event and assign the name to return-symbol

```

```

(if name(ppc-event-map(ppc)) ~= 'none then
  let (next-state : symbol = undefined)
  next-state <-
    find-state(name(defined-name(o)), name(ppc-event-map(ppc)));
  (if defined?(ppc-post-map(ppc)) then
    format (true, " ");
    format (true, "(return-symbol <- '~A", next-state)
  else
    format (true, " (return-symbol <- '~A", next-state)
  )
);

%% add semicolon between transforms (rules) and line feeds
(if ppc ~= last(behavior-ppc-map(o)) then
  format (true, " ");
else
  format (true, ")")
)

); %% ends enum ppc
format (true, " "); return-symbol~%" %% returns name of next state

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%%% Searches for the original name in the OML table and returns the unique
%%% attribute name

function convert-attr-name(o : symbol) =

let (new-name : string = undefined)
new-name <- princ-to-string(o);
(enumerate s over ATTR-NAME-TABLE do
  if s.old-name = new-name then
    new-name <- s.uniq-name);
new-name                                %% return the unique name

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%%% Given the event name declared in a behavior rule, this finds the next state
%%% to transition to.

function find-state(behavior-name: symbol,
  event-name : symbol) =

let (current-state : symbol = undefined,
  next-state : symbol = undefined)

(enumerate entry over descendants-of-class(ROOT,'Relation-Table) do
  if behavior-name = name(to-obj-map(entry)) then
    current-state <- name(from-obj-map(entry))
);
(enumerate entry over descendants-of-class(ROOT,'Relation-Table) do
  if current-state = name(from-obj-map(entry)) and
  event-name = name(assoc-obj-map(entry)) then
    next-state <- name(to-obj-map(entry))

```

```

    );
    next-state

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

%%% Called by Print-Decision-Table, this function determines the type of the
%%% action entry element and prints out the element

function print-action-entry(o : object) =

    if integer-literal(action-value(o)) then
        format(true, " ~D", integer-value-of(action-value(o)))

    elseif real-literal(action-value(o)) then
        format(true, " ~D", real-value-of(action-value(o)))

    elseif false-literal(action-value(o)) then
        format(true, " false")

    elseif true-literal(action-value(o)) then
        format(true, " true")

    elseif string-literal(action-value(o)) then
        format(true, " ~S", string-value-of(action-value(o)))

    elseif identifier-use(name-use(o)) then
        let (temp : string = undefined,
            dot : char = #\.)
        temp <- princ-to-string(name-use(o));
        (if dot in temp then
            format(true, " ~A", convert-attr-name(name(name-use(o))))
        else
            format(true, " '~A", name(name-use(o)))
        )

    elseif arithmetic-expression(action-expr(o)) then
        print-expression(action-expr(o))

    else format (true, " Oh Oh, value type is not defined.")

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

%%% This function translates each state object into a function.
%%% The state space attribute is used to test whether the system's current state
%%% satisfies the state space values. If the state space is valid, then a
%%% list of valid events, event-behaviors, and state behaviors will be created.

function print-state-function (o : object) =
let (valid-str : string = "--% VALID STATE SPACE~%",
    invalid-str : string = "--% INVALID STATE SPACE~%",
    valid-ext-ev : seq(symbol) = [],
    valid-ev-beh : seq(symbol) = [],
    state-beh : seq(symbol) = [])

%%% creates a seq of valid external events for the state and a seq of valid
%%% behaviors associated with each external event. The ith element in each set

```

%% are associated with each other.

```

(enumerate entry over descendants-of-class(ROOT,'Relation-Table) do
  if name(to-obj-map(entry)) = name(defined-name(o)) and
    name(assoc-obj-map(entry)) in EXT-EV-SET then
    valid-ext-ev <- concat(valid-ext-ev, [name(assoc-obj-map(entry))]);
    (enumerate temp over descendants-of-class(ROOT,'Relation-Table) do
      if name(from-obj-map(temp)) = name(assoc-obj-map(entry)) and
        name(assoc-obj-map(temp)) = 'ICO then
        valid-ev-beh <- concat(valid-ev-beh, [name(to-obj-map(temp))])
    )
);

%% print function name and writes the two sequences above into the translated
%% file

format(true, "function ~A() : return-values = ~%", name(defined-name(o)));
format (true, "let (valid-ext-event : seq(symbol) = ~%      [");
(enumerate ext-ev over valid-ext-ev do
  if ext-ev ~= last(valid-ext-ev) then
    format (true, "'~A, ", ext-ev)      %%separate elements with commas
  else
    format (true, "'~A", ext-ev)
);
format (true, "],~%");                  %%close the sequence

format (true, "      valid-event-beh : seq(symbol) = ~%      [");
(enumerate ev-beh over valid-ev-beh do
  if ev-beh ~= last(valid-ev-beh) then
    format (true, "'~A, ", ev-beh)      %%separate elements with commas
  else
    format (true, "'~A", ev-beh)
);
format (true, "],~%");                  %%close the sequence

state-beh <- print-state-behaviors(name(defined-name(o)));
format (true, "      state-beh : seq(symbol) = [");
(enumerate st-beh over state-beh do
  if st-beh ~= last(state-beh) then
    format (true, "'~A, ", st-beh)      %%separate elements with commas
  else
    format (true, "'~A", st-beh)
);
format (true, "],~%");                  %%close the sequence

format (true, "      return-tuple : return-values = undefined)~%~%");

format (true, "      format(true, \"The current state of the system is ~S\");~%",
          name(defined-name(o)));

%% print if statement to check if the state-space is correct

format (true, "      (if ~%      ");
(enumerate expr over set-to-seq(state-space-map(o)) do
  print-expression(expr);
  if expr ~= last(set-to-seq(state-space-map(o))) then

```

```

        format(true, " and~%      ")
    else
        format(true, "~% then~%")
    );

%% if state-space is correct

format (true, "      format(true, ~S);~%", valid-str);
format (true, "      return-tuple <- <'valid, valid-ext-event,
                                valid-event-beh, state-beh>~%";
format(true, " else~%");

%% if state-space is incorrect

format (true, "      format(true, ~S);~%", invalid-str);
format (true, "      return-tuple <- <'invalid, [], ['~A], state-beh>);~%",
                                name(defined-name(o)));

%% send return-tuple back to calling function

format (true, " return-tuple ~%");
format(true, "~%")

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
function print-state-behaviors(state-name: symbol) =

    let (behavior : seq(symbol) = [])

        (enumerate entry over descendants-of-class(ROOT,'Relation-Table) do
            if state-name = name(from-obj-map(entry))
                and name(assoc-obj-map(entry)) = 'IC0 then
                    behavior <- prepend(behavior, name(to-obj-map(entry)))
        );

    behavior      %% return the seq of behaviors associated with the state-name

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

%%% This function translates OML process objects into Refine functions. Each
%%% time the Refine function is called, it will perform one of two major tasks:
%%% 1- check if all inflows are satisfied.
%%% 2- execute process behavior and return info to calling function

function print-proc-function(proc : object) =
    let(proc-name : symbol = name(defined-name(proc)),
        all-inflows : seq(symbol) = [],
        inflow-objs : set(object) = {},
        int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) = [],
        ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) = [],
        next-procs : seq(symbol) = [])

    %% creates a seq of in-flows for the process .

    (enumerate entry over descendants-of-class(ROOT,'Relation-Table) do
        if name(to-obj-map(entry)) = name(defined-name(proc)) and
            name(assoc-obj-map(entry)) ~= 'IC0 then

```

```

    all-inflows <- concat(all-inflows, [(name(assoc-obj-map(entry))))]
  );
%% get flow objects & make a set of their names and types

inflow-objs <- {x | (x:object) flow-object(x) &
                    name(defined-name(x)) in all-inflows};
(enumerate flow over inflow-objs do
  if flow-pp(flow-link-map(flow)) then
    int-flow-set <- append(int-flow-set,
      (<name(flow-data-map(flow)), name(defined-name(flow))>))
  elseif flow-ep(flow-link-map(flow)) then
    ext-flow-set <- append(ext-flow-set,
      (<name(flow-data-map(flow)), name(defined-name(flow))>))
);
%% writes the flow-set sequences above into the translated file

format(true, "function ~A(dowhat : symbol) : return-values = ~%", proc-name);

format (true, "let (int-flow-set : seq(tuple(flow-type : symbol,
                                         flow-name : symbol)) = ~%      [");
(enumerate flow over int-flow-set do
  if flow ~= last(int-flow-set) then
    format (true, "<'~A, '~A>," , flow.flow-type, flow.flow-name)
  else
    format (true, "<'~A, '~A>," , flow.flow-type, flow.flow-name)
);
format (true, "],~%");          %%close the sequence

format (true, "      ext-flow-set : seq(tuple(flow-type : symbol,
                                         flow-name : symbol)) = ~%      [");
(enumerate flow over ext-flow-set do
  if flow ~= last(ext-flow-set) then
    format (true, "<'~A, '~A>," , flow.flow-type, flow.flow-name)
  else
    format (true, "<'~A, '~A>," , flow.flow-type, flow.flow-name)
);
format (true, "],~%");          %%close the sequence

format (true, "      intflows-valid : boolean = false,
      check-flow : object = undefined,
      return-tuple : return-values = <'invalid,[]>~%~%" );

%% check to see if all inflows are valid:
%% gather all the inflow objects related to the process, enumerate
%% over the attributes, checking if they are defined.
%% return-attribute-list function is defined in file obj-utilities.re
%% it expects an object as input & returns a set of refine bindings

format(true, "(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)))));

```





```

%%% This function adds a state simulation function to the translated file.
%%% It is only added when the OML spec has state based information in it.

```

```

function add-state-sim-function(Name : string) =
  format (true, "function sim() =~%");

  format (true, "let (sfunction : return-values = undefined,
    st-name : symbol = '~A, %% assume first state in OML file is initial
    done : boolean = false,
    reply : integer = undefined)~%~%", INITIAL-STATE);

  format (true, "  while ~Adone do~%", tilde);
  format (true, "    sfunction <- funcall(st-name);
    (if sfunction.validity = 'valid then
      reply <- Make-Menu(sfunction.events, \"Events that can occur:\");
      (if Reply <= size(sfunction.events) then
        funcall(sfunction.behaviors(reply));

        enumerate st-beh over sfunction.st-behaviors do
          st-name <- funcall(st-beh)
        elseif Reply = size(sfunction.events)+2 then
          done <- true          %% selects quit
        )
      else
        %% not valid state
        done <- true;~%");
    format (true, "format (true, \"The system's current state space conflicts with
the state space required to be in the above mentioned state. Here are the
current attribute values in the system. Compare them with the required values
specified in your specification to find the inconsistencies.\"A\\%\\%\";~%~%", tilde);
    format (true, "      (enumerate obj over [obj | (obj : ~A) ~A(obj)] do~%",
      Name, Name);
    format (true, "      (enumerate attr over Return-Attribute-List(obj) do~%");
    format (true, "format (true, \"  ~AA.~AA : ~AA~A\\%\\%\", name(obj), name(attr),
retrieve-attribute(obj, attr))~%", tilde,tilde,tilde,tilde);
    format (true, "      )))~%~%")

```

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXX ADD-PROC-SIMULATION-FUNCTION XXXXXXXXXXXXXXXXXXXXXXXXXXXX
%%% This function adds a process simulation function to the translated file.
%%% It is only added when the OML spec has process based information in it.

```

```

function add-proc-sim-func() =
  let(initial-procs : set(symbol) = PROC-NAMES)

  format (true, "function sim() =~%");

  format (true, "let (pfunction : return-values = undefined,
    done : boolean = false,
    reply : integer = undefined,
    test : return-values = undefined,
    valid-procs : seq(symbol) = [],
    init-procs : seq(symbol) = []);

  (enumerate pname over PROC-NAMES do
    (enumerate entry over descendants-of-class(ROOT,'Relation-Table) do
      if pname = name(to-obj-map(entry)) then
        if ex (x) (flow-object(x) &

```

```

        name(defined-name(x)) = name(assoc-obj-map(entry)) &
        ~flow-ep(flow-link-map(x))) then
    initial-procs <- initial-procs less pname
));

(enumerate pname over set-to-seq(initial-procs) do %% prints initial procs
  if pname ~= last(set-to-seq(initial-procs)) then
    format (true, "'~A, ",pname)
  else
    format (true, "'~A", pname) );
format (true, "]"~%"");          %%close the sequence

%% 1- display list of all processes and ask user to select initial process
%% 2- Execute the process. Process returns a tuple <valid, next-procs>
%% 3- If valid then display list of potential next processes to execute.

format (true, "  reply <- Make-Menu(init-procs,
    \"Choose one of these processes to initialize the simulation:\");~%");
format (true, "  (if Reply <= size(init-procs) then
    pfunction <- funcall(init-procs(reply), 'execute);
    while ~Adone do~%", tilde);
format (true, "          valid-procs <- [];
    (if pfunction.validity = 'valid then
      (if size(pfunction.next-procs) > 0 then
        (enumerate proc over pfunction.next-procs do
          test <- funcall(proc, 'check);
          (if test.validity = 'valid then
            valid-procs <- append(valid-procs, proc)));
      reply <- Make-Menu(valid-procs,
        \"Select a process that may potentially execute at this point:\");
      (if Reply <= size(valid-procs) then
        pfunction <- funcall(valid-procs(reply), 'execute)
      elseif Reply = size(valid-procs)+2 then
        done <- true)          %% selects quit
      else pfunction.next-procs <- init-procs)
    else
      %% not valid process
      done <- true))~%")

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
function print-predicate-expression (o : object) =

  if r-equal(o) then
    format(true, " = ");
    print-expression(argument(o))

  elseif r-not-equal(o) then
    format(true, " ~A= ", tilde);
    print-expression(argument(o))

  elseif r-greater-than(o) then
    format(true, " > ");
    print-expression(argument(o))

  elseif r-greater-or-equal(o) then
    format(true, " >= ");

```

```

        print-expression(argument(o))

elseif r-less-than(o) then
    format(true, " < ");
    print-expression(argument(o))

elseif r-less-or-equal(o) then
    format(true, " <= ");
    print-expression(argument(o))

else
    format(true, "not looking at right thing~%")

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
function print-statement (stmt : object) =

    if assignment-statement(stmt) then
        (if ex(x,y,z,w) ( x = name(LHS(stmt)) & (<x,y> in FLOW-NAMES
            & z = name(RHS(stmt)) & (<z,w> in FLOW-NAMES )then
            format(true, "assign-object('~A','~A','~A')~%",
                name(RHS(stmt)), name(LHS(stmt)),
                arb({y | (x : symbol, y : symbol) x = name(RHS(stmt))
                    & (<x,y> in FLOW-NAMES)}))
        else
            format(true, "~A", convert-attr-name(name(LHS(stmt))));
            format(true, " <- ");
            print-expression(RHS(stmt)) )

    elseif display(stmt) then
        (if defined?(name-use(stmt)) then
            format(true, "format(true, \"~A\\\\pp\\\\\",~A)",tilde,
                convert-attr-name(name(name-use(stmt))))
        elseif defined?(display-set(stmt)) then
            format(true, "enumerate element over ~%");
            print-expression(display-set(stmt));
            format(true, " do~%");
            format(true, "    format(true, \"~A\\\\pp\\\\\",element)",tilde)
        )

%      elseif create(stmt) then

%      elseif destroy(stmt) then

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
function print-expression (o : object) =

    if integer-literal(o) then format(true, "~D", integer-value-of(o))

    elseif real-literal(o) then format(true, "~G", real-value-of(o))

    elseif true-literal(o) then format(true, "true")

    elseif false-literal(o) then format(true, "false")

```

```

elseif identifier-use(o) then
  let (temp : string = undefined,
      dot : char = #\.)
  temp <- princ-to-string(name(o));
  (if dot in temp then
    format(true, "~A", convert-attr-name(name(o)))
    elseif name(o) in OBJ-NAMES then    %% is argument an object name?
    format(true, "~A", name(o))
    else
    format(true, "'~A", name(o))
  )
elseif string-literal(o) then format(true, "~S", string-value-of(o))

elseif arithmetic-add(o) then
  format(true, "(");
  print-expression(argument-1(o));
  format(true, " + ");
  print-expression(argument-2(o));
  format(true, ")")

elseif arithmetic-subtract(o) then
  format(true, "(");
  print-expression(argument-1(o));
  format(true, " - ");
  print-expression(argument-2(o));
  format(true, ")")

elseif arithmetic-multiply(o) then
  format(true, "(");
  print-expression(argument-1(o));
  format(true, " * ");
  print-expression(argument-2(o));
  format(true, ")")

elseif arithmetic-divide(o) then
  format(true, "(");
  print-expression(argument-1(o));
  % (if type-integer(type-of-expression(argument-1(o))) then
  %   format(true, " div ")
  %   else format(true, " / ");
  format(true, " / ");
  print-expression(argument-2(o));
  format(true, ")")

elseif unary-minus(o) then
  format(true, "minus(");
  print-expression(argument-1(o));
  format(true, ")")

elseif boolean-and(o) then
  format(true, "(");
  print-expression(argument-1(o));
  format(true, " & ");
  print-expression(argument-2(o));
  format(true, ")")

```

```

elseif boolean-or(o) then
    format(true, "(");
    print-expression(argument-1(o));
    format(true, " or ");
    print-expression(argument-2(o));
    format(true, ")")

elseif boolean-not(o) then
    format(true, "~A(", tilde);
    print-expression(argument-1(o));
    format(true, ")")

elseif compare-equal(o) then
    format(true, "(");
    print-expression(argument-1(o));
    format(true, " = ");
    print-expression(argument-2(o));
    format(true, ")")

elseif compare-not-equal(o) then
    format(true, "(");
    print-expression(argument-1(o));
    format(true, " ~= ", tilde);
    print-expression(argument-2(o));
    format(true, ")")

elseif compare-greater-than(o) then
    format(true, "(");
    print-expression(argument-1(o));
    format(true, " > ");
    print-expression(argument-2(o));
    format(true, ")")

elseif compare-greater-or-equal(o) then
    format(true, "(");
    print-expression(argument-1(o));
    format(true, " >= ");
    print-expression(argument-2(o));
    format(true, ")")

elseif compare-less-than(o) then
    format(true, "(");
    print-expression(argument-1(o));
    format(true, " < ");
    print-expression(argument-2(o));
    format(true, ")")

elseif compare-less-or-equal(o) then
    format(true, "(");
    print-expression(argument-1(o));
    format(true, " <= ");
    print-expression(argument-2(o));
    format(true, ")")

elseif compare-in(o) then

```

```

format(true, "(");
format(true, "~A", name(argument-1(o)));
format(true, " in ");
format(true, "~A", name(argument-2(o)));
format(true, ")")

elseif compare-for-all(o) then
format(true, "(fa (");
(enumerate name over name-uses(o) do
  print-expression(name);
  if name ~= last(name-uses(o)) then
    format(true, ", ")
  );
format(true, ")");
(enumerate expr over set-arg(o) do
  print-expression(expr);
  if expr ~= last(set-to-seq(set-arg(o))) then
    format(true, " &~% ")
  );
format(true, " =>~%");
print-expression(argument-1(o));
format(true, ")")

elseif compare-exists(o) then
format(true, "(ex (");
(enumerate vars over name-uses(o) do
  format(true, "~A", name(vars));
  if vars ~= last(name-uses(o)) then
    format(true, ", ")
  );
format(true, ")");
(enumerate expr over set-arg(o) do
  print-expression(expr);
  if expr ~= last(set-to-seq(set-arg(o))) then
    format(true, " &~% ")
  );
format(true, ")")

elseif set-union(o) then
format(true, "(");
format(true, "~A", name(argument-1(o)));
format(true, " with copy-object(");
print-expression(argument-2(o));
format(true, ")")

elseif set-diff(o) then
format(true, "(");
format(true, "setdiff(");
format(true, "~A, ", name(argument-1(o)));
print-expression(setbuilder-map(o));
format(true, ")");
format(true, ")")

elseif getitem(o) then
format(true, "(");
format(true, "arb(");

```

```

print-expression(setbuilder-map(o));
format(true, ")))"

elseif getset(o) then
  format(true, "(");
  print-expression(setbuilder-map(o));
  format(true, ")))"

elseif setbuilder(o) then
  format(true, "{~A | (~A) ", name(defined-name(o)), name(defined-name(o)));
  (enumerate cond over set-diff-condition(o) do
    print-expression(cond);
    if cond ~= last(set-diff-condition(o)) then
      format(true, " &~% ")
    else
      format(true, "}") )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PRINT-DELETE-OBJECT-FUNCTION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Writes a function that will clear the object base.

function Print-Delete-Object-Function(Name : string) =

  format (true, "function clear-objects() =
(enumerate obj over [obj | (obj : ~A) ~A(obj)] do~%", Name, Name);
  format (true, "      erase-object(obj))~%"

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PRINT-ATTR-TABLE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% This function is used to print the table that holds the original entity names
%% and attribute names and the converted entity names and attribute names. This
%% function is only used for debugging purposes.

function Pmt() =
  enumerate d over ATTR-NAME-TABLE do
    format(true, "OLD: ~A, NEW: ~A~%",
      d.old-name, d.uniq-name)

function Pon() =          %% short for print object name
  enumerate d over OBJ-NAMES do
    format(true, "~A ~%", d)

```

## C.2 Utilities

The some of the following code was developed by Capt Mary Anne Randour is extremely useful for REFINe object manipulation.

```
!! in-package ('RU)
!! in-grammar ('user)

#||
function symbol-2-index(enum-seq : seq(symbol), val : symbol) : integer =
  let (i : integer = 1)
    (while (val ~= enum-seq(i)) and (i < size(enum-seq)) do i <- i + 1);
    (if i > size(enum-seq) then
      format(true, "Error: enumeration symbol not defined"));
    i

function int-2-char(i : integer) : string =
  princ-to-string(code-char(i))

function val (enum-seq : seq(symbol), index : integer) : string =
  princ-to-string(enum-seq(index))
||#
%-----

%% erases Refine database so that ada-trees are not concatenated together
function erase() =
  enumerate o over
  [o | (o : OML-object) OML-object(o)] do
    erase-object(o)

%-----

function oml-root () =
  up-to-root(arb( {w | (w : informal-model)
                    oml-object(w)} ))

%-----

function pf(filename) =
  let(g=find-object('re::grammar, 'oml))
  erase();
  parse-file(filename, false, g, g, find-package("RU"));
  mcu(oml-root())

%-----

function do-rule-search() =
  rs()

%-----
```



#||

File name: menu.re

Author : Capt Mary Anne Randour

Description: Contains functions that will query the user for a selection from a list of possible choices

Rules: None

Functions: Make-Menu

Make-Object-Menu

History:

13 Aug - Baselined

25 Aug - Added a quit line

1 Sep - Added continue line

||#

%-----  
" Given a sequence of symbols and a prompt string, this function displays the prompt, lists all of the symbols in the sequence, and prompts the user for a selection. If the selection is not in the proper range (i.e., between 1 and the size of the sequence), it displays an error message and reprompts for a selection"

function Make-Menu(Menu-Choices : seq(symbol), Prompt : string) : integer =

```
let (Response : integer = size(Menu-Choices) + 3)
(while Response > size(Menu-Choices) + 2 or Response < 1 do
  format(t, "~s ~%", Prompt);
  (enumerate index from 1 to size(Menu-Choices) do
    format(t, "~d ) ~s ~%", index, Menu-Choices(index))
  );
  format(t, "~d ) Continue ~%", size(Menu-Choices)+1);
  format(t, "~d ) Quit ~%", size(Menu-Choices)+2);
  Response <- Read-Integer("");
  if Response > size(Menu-Choices) + 2 or Response < 1 then
    format(t, "Invalid Response ~%")
); % end while
Response
```

%-----  
" Same as above except it takes a list of objects and uses the name to display the choices"

function Make-Object-Menu(Menu-Choices : seq(object),  
Prompt : string) : integer =

```
let (Response : integer = size(Menu-Choices) + 3)
```

```

(while ((Response > (size(Menu-Choices) + 2)) or-else (Response < 1)) do
  format(t, "~s ~%", Prompt);
  (enumerate index from 1 to size(Menu-Choices) do
    format(t, "~d ) ~s ~%", index, name(Menu-Choices(index)))
  );
  format(t, "~d ) Continue ~%", size(Menu-Choices)+1);
  format(t, "~d ) Quit ~%", size(Menu-Choices)+2);
  Response <- Read-Integer("");
  (if Response > size(Menu-Choices) + 2 or-else Response < 1 then
    format(t, "Invalid Response ~%")
  )
); % end while
Response

%-----
function assign-object(copy-from:symbol, copy-to:symbol, object-type: symbol) =

  (enumerate flow-attr over return-attribute-list(find-object(object-type,
                                                             copy-from)) do
    store-attribute(find-object(object-type, copy-to),
                    flow-attr,
                    retrieve-attribute(find-object(object-type, copy-from),
                                       flow-attr))
  )

%-----
% finds users

function fuab (x: set(object)) =
  enumerate user over x do
    format(true, " ~\\pp\\~%", user)

```

```

; File name: lisp-utilities.lisp

; Description: Contains lisp utilities

; Rules:
; None
; Functions:
; Read-Input
; File-Exists
; RD-On
; Rd-Off
; other unused functions

; History:
; Baselined - 13 Aug
;

;-----
;;; reads input, returns a number or a string
(defun read-input-orig ()
  (let* ((input (read-line))
         (stripped (read-from-string input))) ; strips the quotes

    (cond
      ((numberp stripped) stripped)
      (t input) ) ) )

;-----
;;; read the input as above, doesn't bomb if user hits return
;;; Returns either a string or a number
(defun read-input ()
  (let* ((input (read-line))
         (stripped (read-from-string input nil ))) ; strips the quotes,
                                                  ; doesn't return error
                                                  ; if just hit return

    (cond
      ((numberp stripped) stripped)
      (t input) ) ) )

;-----
;;; Tests if the given file-name exists
(defun File-Exists (file-name)
  (cond ( (probe-file file-name) t)
        (t nil))
)

```

```

;;; These functions have been handed down from ???
;-----
;;; redirect *standard-output* to a file, format statements will write
;;; to the specified file. It's turned off by RD-Off
(defun RD-On (file)
  (setq *old-std-output* *standard-output*
        *standard-output* (open file
                                :if-exists      :supersede
                                :if-does-not-exist :create
                                :direction      :output) ))
;-----

;;; redirect *standard-output* to *old-std-output*
(defun RD-Off ()
  (cond ( (streamp *old-std-output*)
          (close *standard-output*)
          (setq *standard-output* *old-std-output*) )
        ('() ) ))

;; Not used yet
;-----
;;; write output to a file

(defun write-report (x file)
  (with-open-file (stream file :direction :output)
    (write-report-to-stream stream x) ))

;-----
;;; redirect *error-output* to a file
(defun Error-RD-On (file)
  (setq *old-err-output* *error-output*
        *error-output* (open file
                                :if-exists      :supersede
                                :if-does-not-exist :create
                                :direction      :output) ))
;-----

;;; redirect *error-output* to *old-err-output*
(defun Error-RD-Off ()
  (cond ( (streamp *old-std-output*)
          (close *error-output*)
          (setq *error-output* *old-err-output*) )
        ('() ) ))
;-----

```

```

;;; redirect *debug-io* to a file
(defun Debug-RD-On (file)
  (setq *old-debug-io* *debug-io*
        *debug-io* (open file
                           :if-exists      :supersede
                           :if-does-not-exist :create
                           :direction      :output) ))

```

```

;-----
;;; redirect *debug-io* to *terminal-io*
(defun Debug-RD-Off ()
  (cond ( (stream-p *old-std-output*)
          (close *debug-io*)
          (setq *debug-io* *old-debug-io*) )
        ('() ) ))

```

```

!! in-package("RU")
!! in-grammar('user)

#||
File name: read-utilities.re

Description: Contains functions that read in different data types. They perform all
type checking so the calling program is guaranteed to get a value of the correct
type. The read with defaults allows the calling program to send a default value.
If the user enters return, this value is returned.

Rules:
None

Functions:
Read-String
Read-Integer
Read-Real
Read-Symbol
Read-Boolean
Read-Any-Type
Read-Yes-Or-No
Read-String-Default
Read-Integer-Default
Read-Real-Default
Read-Symbol-Default
Read-Boolean-Default
Read-Any-Type-Default

History:
13 Aug - Baseline
21 Aug - Changed formats to use ~a instead of ~s so the " are not
displayed as part of the prompts.

||#

var Null-Value : any-type = ""
% what read-input returns if given a carriage return

" Used to tell the valid types that can be read using these functions
This will allow programs to test if a symbol is in Valid-Types so it can build
the function call and use the lisp funcall call to invoke the proper program.
This avoids big if-then-elseif statements "

var Valid-Types : set(symbol) = {'string, 'integer, 'real, 'symbol,
                                'boolean, 'any-type}

%%----- Read Functions-----

% Contains functions to read in data of a specific data type. If the input is
% not valid, it reports the error, and prompts for another value

%-----

```

```

function Read-String(Prompt : string) : string =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  let ( str : any-type = read-input()
    (if lisp::numberp(str) then          % if a number was read,
      str <- lisp::princ-to-string(str)  % convert it to a string
    );
    (while ~lisp::stringp(str) do
      format(t, "~%Invalid input, try again: ");
      str <- read-input()
    );
  str

```

```

%-----
function Read-Integer(Prompt : string) : integer =
  (if ~empty(Prompt) then
    format(t, "~A:", Prompt)
  );
  let ( int : integer = read-input()
    (while ~lisp::integerp(int) do
      format(t, "~%Invalid input, try again: ");
      int <- read-input()
    );
  int

```

```

%-----
function Read-Real(Prompt : string) : real =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  let ( real-num : real = read-input()
    (while ~lisp::floatp(real-num) do
      format(t, "~%Invalid input, try again: ");
      real-num <- read-input()
    );
  real-num

```

```

%-----
function Read-Symbol(Prompt : string) : symbol =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  let ( sym : string = read-input()
    (while ~lisp::stringp(sym) do
      format(t, "~%Invalid input, try again: ");
      sym <- read-input()
    );
    string-to-symbol(lisp::string-upcase(sym), "RU")
  % NOTE: I convert the string to upper case so that it can be compared
  % to other symbols
  % string-to-symbol returns a symbol that is case sensitive
  % (it is quoted by |'s)

```

```

%-----

```

```

function Read-Boolean(Prompt : string) : boolean =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  format(t, "(T/t for true, F/f for false): ");
  let(t-or-f : string = read-input())
  (while ~(t-or-f in {"F", "f", "T", "t"}) do
    format(t, "~%Invalid input, try again: ");
    t-or-f <- read-input()
  );
  t-or-f in {"T", "t"}

%-----
function Read-Any-Type(Prompt : string) : any-type =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  read-input()

%-----
" This function had problems if the user entered a number. I now keep
  reading new input if I read a number. I tried using stringp(y-or-no)
  and-then y-or-no in {...}, but that didn't help"
#||
function Read-Yes-Or-No(Prompt : string) : boolean =
  % Returns true if user enters y or Y, false if user enters n or N
  (if ~empty(Prompt) then
    format(t, "~A ", Prompt)
  );
  let (y-or-no : string = read-input())
  (while lisp::numberp(y-or-no) do
    format(t, "~%Invalid input, try again: ");
    y-or-no <- read-input()
  );
  (while y-or-no ~in {"y", "Y", "n", "N"} do
    format(t, "~%Invalid input, try again: ");
    y-or-no <- read-input();
    (while lisp::numberp(y-or-no) do
      format(t, "~%Invalid input, try again: ");
      y-or-no <- read-input()
    )
  );
  y-or-no in {"Y", "y"}
||#

% Here's another way to do this
function Read-Yes-Or-No(Prompt : string) : boolean =
  lisp::y-or-n-p(prompt)

%%----- Read Functions With Defaults-----
%% Read functions that allow for a default value

```



```

%-----
function Read-String-Default(Prompt : string, Default : string) : string =
  (if ~empty(Prompt) then
    format(t, "~A", Prompt)
  );
  format(t, " (~A): ", default);
  let ( str : any-type = read-input()
    (if lisp::numberp(str) then          % if a number was read,
      str <- lisp::princ-to-string(str)  % convert it to a string
    );
    (while ~stringp(str) and ~(lisp::equal(str, Null-Value)) do
      format(t, "~%Invalid input, try again: ");
      str <- read-input()
    );
    if lisp::equal(str, Null-value) then
      Default
    else
      str

  % For reading in integers and real numbers, read in as a string (so it can be
  % compared to the null-value and check if the string is really an integer
  % or real number (using the read-from-string function). Read-from-string
  % returns two values, the first is what's in the string, the second is the index
  % of the first character NOT read.

```

```

%-----
function Read-Integer-Default(Prompt : string, Default : integer) : integer =
  (if ~empty(Prompt) then
    format(t, "~A:", Prompt)
  );
  format(t, " (~d): ", default);
  let ( int : string = lisp::read-line()
    (while ~lisp::integerp(lisp::read-from-string(int)) and int ~= Null-Value do
      format(t, "~%Invalid input, try again: ");
      int <- lisp::read-line()
    );
    if int = Null-value then
      Default
    else
      lisp::read-from-string(int)

```

```

%-----
function Read-Real-Default(Prompt : string, Default : real) : real =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  format(t, " (~d): ", default);
  let ( real-num : string = lisp::read-line()
    (while ~lisp::floatp(lisp::read-from-string(real-num)) and real-num ~= Null-Value do
      format(t, "~%Invalid input, try again: ");
      real-num <- lisp::read-line()
    );
    if real-num = Null-value then
      Default
    else

```

```

lisp::read-from-string(real-num)

%-----
function Read-Symbol-Default(Prompt : string, Default : symbol) : symbol =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  format(t, " (~A): ", default);
  let ( sym : string = read-input())
    (while ~lisp::stringp(sym) and sym ~= Null-Value do
      format(t, "~%Invalid input, try again: ");
      sym <- read-input()
    );
    if sym = Null-value then
      Default
    else
      string-to-symbol(lisp::string-upcase(sym), "RU")
  % NOTE: I convert the string to upper case so that it can be compared to
  % other symbols string-to-symbol returns a symbol that is case sensitive
  % (it is quoted by |'s)

%-----
function Read-Boolean-Default(Prompt : string, Default : boolean) : boolean =
  (if ~empty(Prompt) then
    format(t, "~A ", Prompt)
  );
  format(t, "(T/t for true, F/f for false:) ");
  format(t, " (~A): ", default);
  let(t-or-f : string = read-input())
    (while ~(t-or-f in {"F", "f", "T", "t"}) and t-or-f ~= Null-Value do
      format(t, "~%Invalid input, try again: ");
      t-or-f <- read-input()
    );
    if t-or-f = Null-value then
      Default
    else
      t-or-f in {"T", "t"}

%-----
function Read-Any-Type-Default(Prompt : string, Default : any-type) : any-type =
  (if ~empty(Prompt) then
    format(t, "~A: ", Prompt)
  );
  format(t, " (~A): ", default);
  let (ans : any-type = read-input())

  if lisp::equal(ans, Null-value) then % use the lisp::equal incase of strings
    Default
  else
    ans

```

```
!! in-package("RU")
!! in-grammar('user)
```

```
#||
```

```
File name: obj-utilities.re
```

```
Description: This file contains functions useful when manipulating objects,
regardless of the domain model being used.
```

```
Rules:
```

```
none
```

```
Functions:
```

```
Return-Attribute-List
```

```
Tell-Set-Seq-Type
```

```
Tell-Set-Seq-Binding
```

```
Get-Attribute-List (For testing and debugging)
```

```
Tell-Type
```

```
Copy-Object
```

```
History:
```

```
Baselined - 13 Aug
```

```
||#
```

```
%-----
```

```
" The type-map gives a conversion from the Refine representation of
data types to simple symbols (Note: strings are handled differently so
they aren't in this list"
```

```
var Type-Map : map(symbol, symbol) =
{| 're::powerset-op      -> 'set,
  're::powersequence-op -> 'seq,
  're::symbol-op        -> 'symbol,
  're::real-op          -> 'real,
  're::integer-op       -> 'integer,
  're::boolean-op       -> 'boolean,
  're::any-type-op      -> 'any-type|}
```

```
%-----
```

```
" REFINES attributes"
```

```
var predefined-attributes : set(symbol) =
{'RE::--TOP-LEVEL-PREPENDUM--,
 'RE::STORED-PROPERTIES,
 'RE::BROWSER-MENU-STRING-FOR-NAMED-OBJECT--SAME-PACKAGE,
 'RE::BROWSER-MENU-STRING-FOR-NAMED-OBJECT--OTHER-PACKAGE,
 'RE::ORDERED-CHILDREN-ATTRIBUTES,
 'RE::CONSTRUCTION-FUNCTION-ATTRIBUTE-ARGS,
 'RE::CONSTRUCTION-FUNCTION,
 'RE::REFINE-INTERNAL?,
 'RE::QUOTED?,
 'RE::LISP-GETFN,
 'RE::LISP-INITIALIZE,
 'RE::LISP-FUNCTION,
 'RE::ALREADY-WARNED-ABOUT,
```

```

'RE::SUBPART-OF,
'RE::SUBPARTS,
'RE::COMPILATIONS,
'RE::ZL-DOCUMENTATION,
'RE::USED-BY,
'RE::MENTIONED-BY,
'RE::DATA-TYPE,
'RE::CHILDREN-ENVIRONMENTS,
'RE::PARENT-ENVIRONMENT,
'RE::COPY-OF,
'RE::BINDING-VALUE-OF,
'RE::PARENT-LINK-NAME,
'RE::PARENT-LINK,
'RE::CLASS,
'RE::ELEMENT-OF,
'RE::PROPS-FROM-READER}

%-----
" Given an object, returns a set of bindings that represent the attributes
  removing the predefined attributes"

function Return-Attribute-List(Obj : object) : set(re::binding) =
  {attrs | (attrs (attrs in class-attributes(instance-of(obj), true)) &
    ~(name(attrs) in predefined-attributes))}

%-----
" Returns all of the subclasses of an object, NOT including the original
  object class"

function Get-SubNodes(obj-type: re::binding) : set(re::binding) =
  let (tempset : set(re::binding) = class-subclasses(obj-type, false) )
  tempset less obj-type % don't include the original object type

%-----
" This function determines the type of the set/seq attribute. If the type
  is an object, it returns the object class name"

function Tell-Set-Seq-Type(attr : re::binding) : symbol =
  let (its-type : object = re::base(re::range-type(re::data-type(attr))))
  if re::class(its-type) = 're::binding-ref then
    re::bindingname(its-type) % returns 'string or '(object-type)
  else
    Type-Map(re::class(its-type))

%-----
" Returns the type of a set or sequence as a binding (assumes its
  some object type)"

function Tell-Set-Seq-Binding(attr : re::binding) : re::binding =
  re::ref-to(re::base(re::range-type(re::data-type(attr))))

```

```

%-----
" displays all of the user-defined attributes and their types.  If want to see
all the attributes, don't comment any lines, if only want to see the
user-defined attributes, comment out the line: if ~(name(atr) in
predefined-attributes) then"

function Get-attribute-list(obj : object) =
let (attr-list : set(re::binding) = class-attributes(instance-of(obj), true))
format(t, "attributes are: ~%");
enumerate atr over attr-list do
% if ~(name(atr) in predefined-attributes) then
format(t, "attr: ~s type ~s ~%", atr, tell-type(atr))

%-----
" Goes through the abstract syntax tree for the representation of the attribute
to find out the attribute's data type.  Since all attributes are maps, we need
to look at the range-type of the data-type.  Both objects and strings have the
same representation at this level so there's a special test for those.  Other-
wise, it uses the Type-Map to translate the type to a simpler-form."

function Tell-Type(attr : re::binding) : symbol =
let (its-type : object = re::range-type(re::data-type(attr)))

if re::class(its-type) = 're::binding-ref then
if defined?(re::bindingname(its-type)) and-then
re::bindingname(its-type) = 'string then
'string
else
'object
else
Type-Map(re::class(its-type))

%-----
" makes a copy of an object, the calling routine must name the new object
used instead of copy-term because copy-term cannot be used with unique names
classes see refine manual pg 3-194.  An alternative could be to undefine the
name, then copy it.  This is a problem if the object contains any named
objects.  This function handles any case (that I can think of)"

function Copy-Object(from-obj : object) : object =

let (Attrs : set(re::binding) = Return-Attribute-List(from-obj),
To-Obj : object = make-object(name(instance-of(from-obj))))

(enumerate attrib over Attrs do

% if it's an object, copy the object and assign the new one
% to the attribute
(if tell-type(attrib) = 'object then
let (sub-obj : object =
Copy-Object(retrieve-attribute(from-obj, attrib)))
store-attribute(to-obj, attrib, sub-obj)

```

```

elseif tell-type(attrib) = 'set then

let (temp-set : set(any-type) = {})
% if it is a set of objects, copy each object,
% otherwise copy the set
(if defined? (Find-Object-Class(Tell-Set-Seq-Type(Attrib))) then %object
  enumerate Set-Item over Retrieve-Attribute(From-Obj, Attrib) do
    Temp-Set <- Temp-Set with Copy-Object(Set-Item)
  else
    Temp-Set <- Retrieve-Attribute(From-Obj, Attrib)
);
Store-Attribute(To-Obj, Attrib, Temp-Set)

elseif tell-type(attrib) = 'seq then

let (temp-seq : seq(any-type) = [])
% if it is a seq of objects, copy each object,
% otherwise copy the seq
(if defined? (Find-Object-Class(tell-set-seq-type(attrib))) then
  %= object
  enumerate seq-item over retrieve-attribute(from-obj, attrib) do
    temp-seq <- append(temp-seq, copy-object(seq-item))
  else
    temp-seq <- retrieve-attribute(from-obj, attrib)
);
store-attribute(to-obj, attrib, temp-seq)

else %not a set, seq, or object

  store-attribute(to-obj, attrib, retrieve-attribute(from-obj, attrib))
) % end if
); %end enumerate

To-Obj % return the object

```

```

!! in-package("RU")
!! in-grammar('user)

%% This object class is defined here as an initialization. The actual OML-obj
%% is further defined in the translated executable OML specification.

var OML-Obj : object-class subtype-of user-object

var Changes-Made : boolean = false

%%var Spec-Parts : map(OML-Obj, seq  ) = {}

#||
File name: modify-obj.re

Description:
The functions in this file allow the user to modify objects - delete
objects, edit existing objects, and add new objects

Rules:
Edit-An-Object
Add-An-Object
Delete-An-Object

Functions:
Modify-Some-Object
Modify-Object
Update-Attr
Find-Subnode
Make-New-Object
Read-Set
Read-Seq
Is-Valid-New-Type
Add-Object
Delete-Object

History:
Baselined 13 Aug
||#

%% Rules that can be performed by the user

rule Edit-An-Object(X: object)
    % Do not change the name to edit-object, it exists already)
    true --> Modify-Some-Object(x)

rule Add-An-Object(X: Object)
    true --> Add-Object(x)

rule Delete-An-Object(X: Object)
    true --> Delete-Object(X)

%-----
" Asks the user for the name of the object, checks that it is a subclass of
component object (i.e., its a subsystem or primitive domain object), and then

```

Modifies the object"

```
function Modify-Some-Object(X : object) =
```

```
  let (obj-to-edit : symbol =
      Read-Symbol-Default("Enter the name of the object to edit", Name(x)))

  let (Edit-Obj : object = find-object('OML-Obj, obj-to-edit))
  if defined?(Edit-Obj) then
    Changes-Made <- true;
    Edit-Obj <- Modify-Object (Edit-Obj)
  else
    format(t, "Object ~s is not a current object that can be edited~%", obj-to-edit)
```

```
%-----
" Given an object, goes through each attribute that's not one of the predefined
attributes and gets a value for it"
```

```
function Modify-Object (obj : object) : object =
  (enumerate atr over return-attribute-list(obj) do
    update-attr(obj, atr)
  );
Obj
```

```
%-----
" Given an object and an attribute, finds the data type of the attribute,
calls the appropriate read function, and stores the value in the attribute.
If the attribute is an object, it first creates an object of that type
(re::bindingname(re::range-type(re::data-type(attribute-binding)))) and
then gets the information for that object "
```

```
function Update-Attr(for-obj : object, attrib : re::binding) =
```

```
  let (attr-type : symbol = tell-type( attrib),
      prompt : string = concat("Enter ", symbol-to-string(name(attrib))),
      current-value : any-type = Retrieve-Attribute(for-obj, attrib) )

  if attr-type = 'real then
    store-attribute(for-obj, attrib, read-real-default(prompt, current-value))
  elseif attr-type = 'integer then
    store-attribute(for-obj, attrib, read-integer-default(prompt, current-value))
  elseif attr-type = 'string then
    store-attribute(for-obj, attrib, read-string-default(prompt, current-value))
  elseif attr-type = 'boolean then
    store-attribute(for-obj, attrib, read-boolean-default(prompt, current-value))
  elseif attr-type = 'symbol then
    store-attribute(for-obj, attrib, read-symbol-default(prompt, current-value))
  elseif attr-type = 'any-type then
    store-attribute(for-obj, attrib, read-any-type-default(prompt, current-value))
  elseif attr-type = 'object then
    if defined?(current-value) then %object already exists, just update it
      store-attribute(for-obj, attrib, Modify-Object(current-value))
    else
```



```

        store-attribute(for-obj, attrib,
            Make-New-Object(re::ref-to(re::range-type(re::data-type(attrib))))))
    elseif attr-type = 'seq then
        format(t, "~s~%", prompt); % Read-Seq doesn't print this prompt
        store-attribute(for-obj, attrib, read-seq(attrib, current-value) )
    elseif attr-type = 'set then
        format(t, "~s~%", prompt); % Read-Set doesn't print this prompt
        store-attribute(for-obj, attrib, read-set(attrib, current-value) )
    else
        format(t, "Unrecognized type ~s ~%", attr-type)

#||
%-----
"Sets the spec object to be the parent of the new object
could instead have a rule:
true --> Parent-Expr(Kid) = parent & kid in spec-parts(parent)"

function Set-To-Parent(Kid, Parent : object) =
    Spec-Parts(Parent) <- append(Spec-Parts(Parent), Kid)

%-----
" Removes the kid from the parent object"

function Remove-From-Parent(Kid, Parent : object) =
    Spec-Parts(Parent) <-
        [objs | (objs : object) objs in Spec-Parts(Parent) &
            name(objs) ~= Name(Kid)] %Remove from application

||#
%-----
" Finds all of the subclasses of of an attribute and if more than one exists,
asks the user which one he wants. Class-Subclasses returns the current class"
#||
function Find-SubNode (attrib : re::binding) : re::binding =

    % first, get the right object class (it may have subclasses)
    let (subnodes : seq(re::binding) =
        set-to-seq(Class-Subclasses(attrib, false) less attrib))
        % remove the current class (attrib) from the list of all subclasses
    let (Object-wanted : re::binding = re::*undefined*)
        % the type of object to create

    (if SubNodes = nil then
        % if it doesn't have any subtypes, the set is nil
        Object-wanted <- attrib
    elseif size(subnodes) > 1 then
        % it has subobject types, find out which one to use

        let (response : integer =
            Make-Object-Menu(subnodes, "Enter which type of object you want to build"))

        Object-wanted <- subnodes(response)

    else % there's only one subtype of object, this probably shouldn't happen
        Object-wanted <- subnodes(1)

```

```

);
(let (subsubnodes : set(re::binding) = Class-Subclasses(object-wanted, false))
  if subsubnodes ~= nil and-then size(subsubnodes) > 1 then
    % The object selected has subnodes, find the object class at this level
    object-wanted <- find-subnode(object-wanted)
  );
object-wanted

" Given an attribute that represents an object, creates an object of that type and
gets all of the attribute data "

||#

function Make-New-Object( attrib : re::binding) : object =

  let (temp-obj : object = make-object(name(Find-Subnode(attrib))))
  Temp-Obj <- Modify-Object(temp-obj);
  Temp-obj

" Reads in a group of items of the given type and puts them into a set. Since a set
may already exist, it first asks if the user wants to change the original set"

function Read-Set(attr : re::binding, current-set : set(any-type)) : set(any-type) =

  let (change : boolean = Read-Yes-Or-No("Do you want to change the current set?"))
  if ~change then
    current-set % return the current value
  else
    let (temp-set : set(any-type) = {},
        of-type : symbol = Tell-Set-Seq-Type(attr))

    format(t, "creating a set of type ~s ~%", of-type);
    (while Read-Yes-Or-No( "Add another element? " ) do
      if of-type = 'integer then
        temp-set <- temp-set with Read-Integer("(an integer)")
      elseif of-type = 'real then
        temp-set <- temp-set with Read-Real("(a real number)")
      elseif of-type = 'string then
        temp-set <- temp-set with Read-String("(a string)")
      elseif of-type = 'symbol then
        temp-set <- temp-set with Read-Symbol("(a symbol)")
      elseif of-type = 'boolean then
        temp-set <- temp-set with Read-Boolean("(a boolean)")
      elseif of-type = 'any-type then
        temp-set <- temp-set with Read-Any-Type("(any-type)")
      else % must be an obj
        temp-set <- temp-set with Make-New-Object(Tell-Set-Seq-Binding(attr))

    ); % end while
    temp-set

```

" Reads in a group of items of the given type and puts them into a sequence. Since a sequence may already exist, it first asks if the user wants to change the original sequence"

```
function Read-Seq(attr : re::binding, current-seq : seq(any-type)) : seq(any-type) =

  let (change : boolean = Read-Yes-Or-No( "Do you want to change the current sequence?"))
  if ~change then
    current-seq % return the current value
  else

    let (temp-seq : seq(any-type) = [],
        of-type : symbol = Tell-Set-Seq-Type(attr))

    format(t, "creating a seq of type ~s ~%", of-type);
    (while Read-Yes-Or-No("Add another element? ") do

      if of-type = 'integer then
        temp-seq <- append(temp-seq, Read-Integer("(an integer)"))
      elseif of-type = 'real then
        temp-seq <- append(temp-seq, Read-Real("(a real)"))
      elseif of-type = 'string then
        temp-seq <- append(temp-seq, Read-String("(a string)"))
      elseif of-type = 'symbol then
        temp-seq <- append(temp-seq, Read-Symbol("(a symbol)"))
      elseif of-type = 'boolean then
        temp-seq <- append(temp-seq, Read-Boolean("(a boolean)"))
      elseif of-type = 'any-type then
        temp-seq <- append(temp-seq, Read-Any-Type("(any-type)"))
      else % must be an object
        temp-seq <- append(temp-seq, Make-New-Object(Tell-Set-Seq-Binding(attr)))

    ); %end while
  temp-seq
```

%%% Functions for adding new objects:

%-----

" The new object type must be a subclass of OML-Obj."

```
function Is-Valid-New-Type (Obj-Type : symbol) =
  Find-Object-Class(Obj-Type) in
  Class-Subclasses(Find-Object-Class('OML-Obj), true)
```

%-----

" Asks for the name of the application for which the new object is to be a part, if the application exists, it then asks for a new object name. It checks that the object name does not exist. It then asks for the type of object to be built, if it is a valid type, it builds a new object, gets the data, and assigns it to the application."

```
function Add-Object (X : object) =
  let (Applic-Name : symbol = Read-Symbol-Default("Enter the application name", name(x)))
```

```

if defined?(find-object('OML-Obj, applic-name)) then
  let (obj-type : symbol = Read-Symbol("What type of object do you want to create?"))

  if Is-Valid-New-Type(Obj-Type) then
    let (Obj-Name : symbol = Read-Symbol("What is the object's name?"))

    if undefined?(find-object('OML-Obj, Obj-Name)) then
      let (New-Obj : object = Modify-Object(make-object(Obj-Type)))
      Changes-Made <- true;
      Set-Attrs(New-Obj, 'name, Obj-Name)    %%;
      Set-To-Parent(New-Obj, find-object('OML-Obj, Applic-name))
%
    else
      format(t, "An object named ~s already exists~%", Obj-Name)

  else
    format(t, "~s is not a valid object type~%", obj-type)

else
  format(t, "Application ~s does not exist in the object base.~%", applic-name)

%%%% Function for erasing  objects:
%-----
" Asks for the object to be deleted, checks that the object exists in the object base,
then asks if the user is sure he wants to erase that object, if he answers yes, the
object is removed from the application definition, and erased"

function Delete-Object (A : object) =
  let (Obj-Name : symbol = Read-Symbol("What object's do you want to delete?"))
  let (Obj : object = Find-Object('OML-Obj, Obj-Name))
  if defined?(Obj) then
    (if Read-Yes-Or-No(concat("Are you sure you want to detete ",
                               Symbol-To-String(Obj-Name), " " )) then
      Changes-Made <- true;
%      Remove-From-Parent(Obj, Parent-Expr(Obj));
      erase-Object(Obj)
    )
  else
    format(t, "~s is not in the object base~%", Obj-Name)

```

## *Appendix D. Home Heater Problem*

### *D.1 Heater Problem Analysis*

The Home Heating System problem comes from the problem set for the Fourth International Workshop on Software Specification and Design, and is based on a problem by S. White presented to 1984 Embedded Computer System Requirement Workshop.

### *D.2 Problem Statement*

"The controller of an oil hot water home heating system regulates in-flow of heat, by turning the furnace on and off, and monitors the status of combustion and fuel flow of the furnace system, provided the master switch is set to "heat" position. The controller activates the furnace whenever the home temperature,  $t$ , falls below  $t_r - 2$  degrees, where  $t_r$  is the desired temperature set by the user. The activation procedure is as follows:

1. the controller signals the motor to be activated;
2. the controller monitors the motor speed and once the speed is adequate it signals the ignition and oil valve to be activated.
3. the controller monitors the water temperature and once the temperature is reached a pre-defined value it signals the circulation valve to be opened. The heated water then starts to circulate through the house.
4. a fuel flow indicator and an optical combustion sensor signal the controller if abnormalities occur. In this case the controller signals the system to be shut off.
5. once the home temperature reaches  $t_r + 2$  degrees, the controller deactivates the furnace by first closing the oil valve and then, after 5 seconds, stopping the motor.

In addition the system is subject to the following constraints:

1. minimum time for furnace restart after prior operation is 5 minutes.

2. furnace turn-off shall be indicated within 5 seconds of master switch shut off or fuel flow shut off. (18)"

### D.3 Entity Relationship Model

The ERM in Figure 23 shows the entities required to specify the home heater. The controller

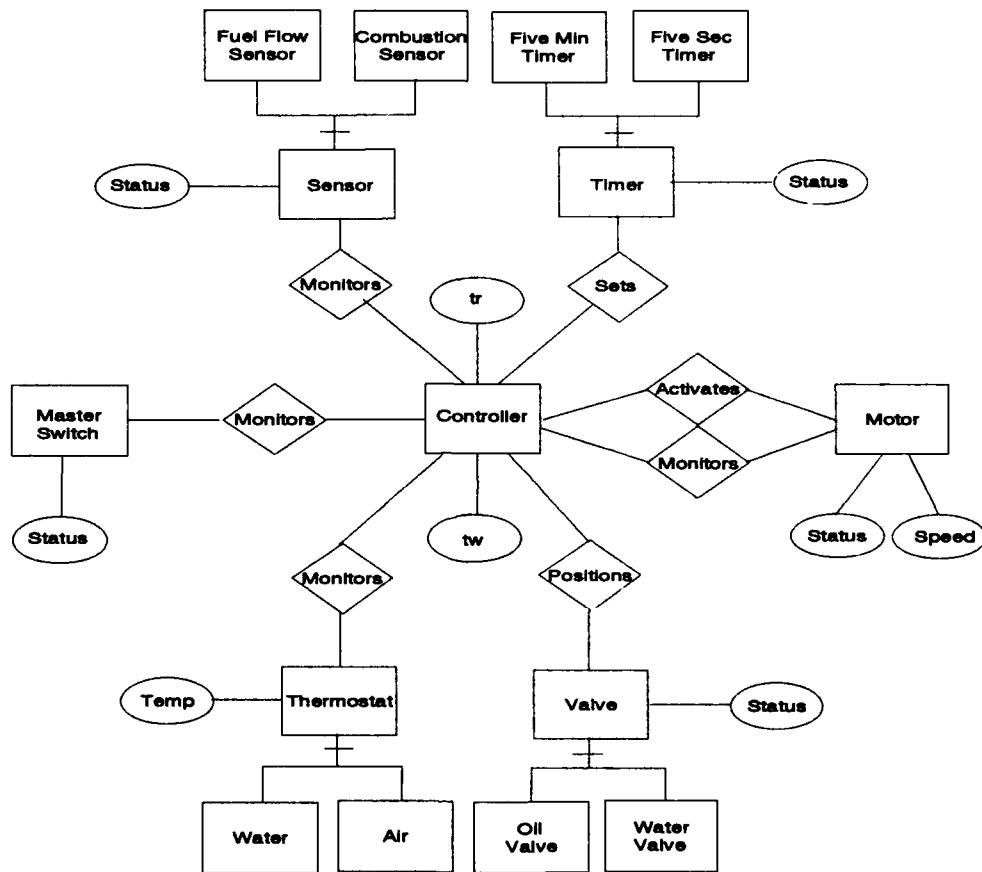


Figure 23. Home Heating System: Entity Relationship Model

has two attributes,  $t_r$ , the desired air temperature, and  $t_w$ , the water temperature that must be reached before water from the boiler will circulate throughout the heater system. The controller monitors certain entities and changes the settings on others. Note that the master switch, the thermostats, and the sensors are external entities, and that their values cannot be changed by the controller. The meaning of the status attribute depends on the entity that it is associated with.

The status of a sensor is either safe or unsafe. The status of the master switch is either “off” or “heat”, and valve’s status is either “open” or “closed”. The status of both the timer and motor is either “off” or “on”.

#### D.4 State Transition Model

This problem easily fits a classical state transition model. The problem statement gives an activation procedure that shows conditions that must be met before the system advances to the next state. Figure 24 depicts the activation and shutdown procedures developed from the problem statement by Blankenship (6:Appendix C). The diagram has been modified to redirect events

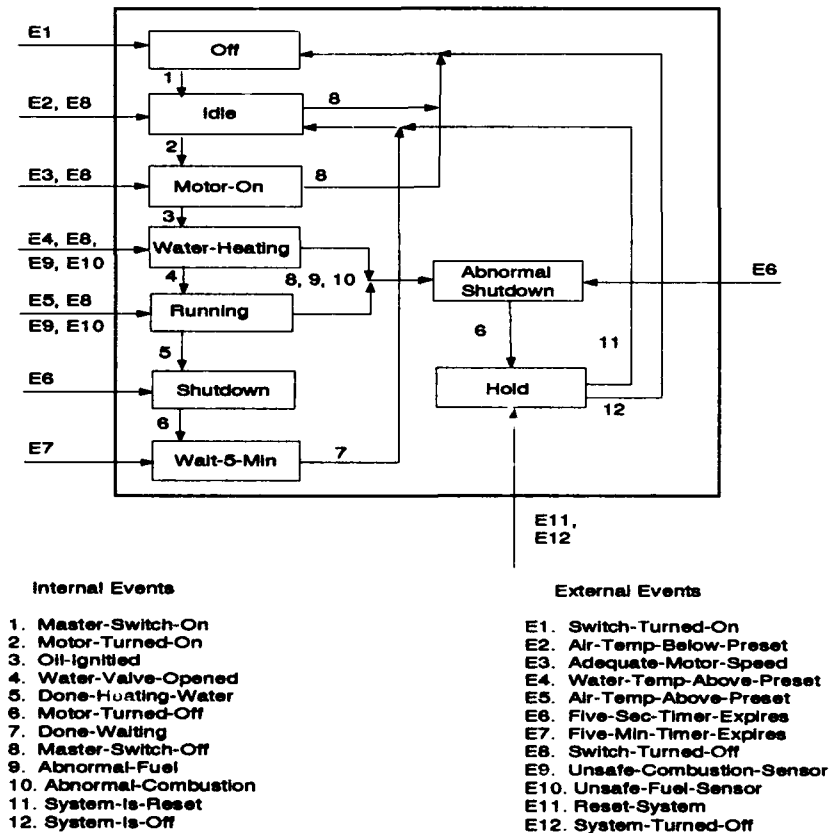


Figure 24. Home Heating System: State Transition Model

dealing with error conditions. The system waits in each state for changes in the environment that trigger transitions to the next state.

Nine states were chosen to model this problem:

**off:** heating system is not on

**idle:** heating system is on, but it is not heating

**motor-on:** (step one of the activation process) water pump (motor) activated

**water-heating:** (step two of the activation process) water pump speed is adequate, fuel ignited, oil valve open

**running:** (final step of the activation process) activation complete, house being heated

**shutdown:** house temperature is sufficient, shut down the system

**wait-5-min:** constraint one: minimum cycle time

**abnormal-shutdown:** an error event has occurred and the system is beginning to shut down

**hold:** a state where the system waits for error correction or for the system to be turned off

The operation of the home heating system can be described by the nine states listed above. Initially, the controller is in the OFF state. When the master switch is placed in the "on" position (Event E1), the controller transitions to the IDLE state (Event 1). There are two possible transitions out of the IDLE state:

1. Should the master switch be returned to the "off" position (Event E8), the controller will transition to the OFF state (Event 8).
2. Should the air temperature drop to two degrees below the desired temperature (Event E2), the controller will transition to the MOTOR-ON state (Event 2).

In the MOTOR-ON state, the water circulation pump is activated. The system will remain in this state until the pump's speed is adequate (Event E3), then the controller will signal the ignition,



open the oil valve, and transition to the WATER-HEATING state (Event 3). Should the master switch be placed in the "off" position (Event E8) while the controller is in the MOTOR-ON state, the pump will be deactivated and the controller will transition to the OFF state (Event 8).

In the WATER-HEATING state, the controller waits for one of four events:

1. When the water temperature reaches the preset temperature (Event E4), the controller will open the water valve, and transition into the RUNNING state (Event 4).
2. If the master switch is returned to the "off" position (Event E8), the controller will close the water and oil valves, start the five second timer, and transition into the ABNORMAL SHUTDOWN state (Event 8).
3. Should the fuel flow sensor detect an abnormal condition (Event E9), the controller will close the water and oil valves, start the five second timer, and transition into the ABNORMAL SHUTDOWN state (Event 9).
4. Should the combustion sensor detect an abnormal condition (Event E10), the controller will close the water and oil valves, start the five second timer, and transition into the ABNORMAL SHUTDOWN state (Event 10).

In the RUNNING state, the controller waits for one of four events:

1. When the air temperature is two degrees greater than the desired temperature, (Event E5), the controller will close the oil valve, start the five second timer, and transition into the SHUTDOWN state (Event 5).
2. If the master switch is returned to the "off" position (Event E8), the controller will transition to the ABNORMAL SHUTDOWN state (Event 8).
3. Should the fuel flow sensor detect an abnormal condition (Event E9), the controller will close the water and oil valves, start the five second timer, and transition into the ABNORMAL SHUTDOWN state (Event 9).

4. Should the combustion sensor detect an abnormal condition (Event E10), the controller will close the water and oil valves, start the five second timer, and transition into the ABNORMAL SHUTDOWN state (Event 10).

The ABNORMAL SHUTDOWN state waits for the five second timer to expire (Event E6), and then shuts the water pump motor off, closes the water valve, turns the ignition off, and transitions to the HOLD state (Event 6).

The SHUTDOWN state waits for the five second timer to expire (Event E6), and then shuts the water pump motor off, closes the water valve, and turns the ignition off. The controller starts the five minute timer and transitions to the WAIT-5-MINUTES state (Event 6).

The WAIT-5-MINUTES state keeps the heater system from entering another heating cycle for five minutes. When the timer expires (Event E7), the controller transitions into the IDLE state (Event 7).

The HOLD state is entered when the heater system has been shut down because of an unsafe sensor reading or because the master switch was turned off during the heating cycle. The controller will remain in this state until the system is reset (Event E11) when it will transition into the IDLE state (Event 11). If the master switch is turned off (Event E12), the controller will transition into the OFF state (Event 12). The HOLD state was included to prevent the controller from transitioning through the first three activation states before sensing an unsafe condition that may not have been corrected.

#### D.5 Heater Problem OML Specification

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% File-Name : h.spec (Home-Heater Specification)
%%%
%%% Authors : Capt Mary Boom, Capt Brad Mallare
%%%
%%% Purpose : OML specification for the home heater problem.
%%%
%%% Unified Abstract Model Components :
%%%   Entities, Relationships, States, Events, Behaviors, and
%%%   Relation-Tables
%%%
%%% Operation : After loading the translation code (trans-oml.fasl4) and
%%% all the other code that it is dependent on, this OML specification
%%% can be translated into an executable specification by typing the
%%% following command at the Refine prompt:
%%%
%%%           (convert "<your-OML-file-name>")
%%%
%%% The name of the generated executable specification will be displayed
%%% on the screen. Additionally, the executable specification will be
%%% automatically compiled and loaded.
%%%
%%% After this file is translated, compiled, and loaded into Refine,
%%% it can be executed by typing the following command at the Refine
%%% prompt:
%%%           (sim)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

specification home-heater

#### ENTITIES

SENSOR class-of entity

type : external

parts

status : symbol range {safe, unsafe}

FUEL-SENSOR instance-of SENSOR

values

status : safe

COMBUSTION-SENSOR instance-of SENSOR

values

status : safe

VALVE class-of entity

type : external

parts

```

    status : symbol range {open, closed}

WATER-VALVE instance-of VALVE
  values
    status : closed

OIL-VALVE instance-of VALVE
  values
    status : closed

TIMER class-of entity
  type : external
  parts
    status : symbol range {off, on}

FIVE-MIN-TIMER instance-of TIMER
  values
    status : off

FIVE-SEC-TIMER instance-of TIMER
  values
    status : off

THERMOSTAT class-of entity
  type : external
  parts
    temp : integer range {0 .. 280}

AIR instance-of THERMOSTAT
  values
    temp : 60

WATER instance-of THERMOSTAT
  values
    temp : 60

MASTER-SWITCH instance-of entity
  type : external
  parts
    status : symbol range {on, off} init-val off

MOTOR instance-of entity
  type : external
  parts
    status : symbol range {on, off} init-val off;
    speed : symbol range {adequate, inadequate} init-val inadequate

IGNITION instance-of entity
  type : external
  parts
    status : symbol range {on, off} init-val off

```

CONTROLLER instance-of entity

type : internal

parts

tr : integer range {32 .. 130} init-val 70; %% preset air temp

tw : integer range {32 .. 280} init-val 180 %% preset water temp

%%%% RELATIONSHIPS %%%%

ACTIVATES instance-of relationship

type : general

cardinality : 1-1

MONITORS instance-of relationship

type : general

cardinality : 1-1

POSITIONS instance-of relationship

type : general

cardinality : 1-1

SETS instance-of relationship

type : general

cardinality : 1-1

%%%% STATES %%%%

%% The first state in an OML spec is assumed to be the initial state

OFF instance-of state

state-space : master-switch.status = off

IDLE instance-of state

state-space : master-switch.status = on;

five-min-timer.status = off;

five-sec-timer.status = off

MOTOR-ON instance-of state

state-space : master-switch.status = on;

motor.status = on;

motor.speed = inadequate;

air.temp < controller.tr - 2;

ignition.status = off;

oil-valve.status = closed

WATER-HEATING instance-of state

state-space : master-switch.status = on;

air.temp < controller.tr + 2;

motor.status = on;

motor.speed = adequate;

```

water.temp < controller.tw;
fuel-sensor.status = safe;
combustion-sensor.status = safe;
water-valve.status = closed;
oil-valve.status = open

```

#### RUNNING instance-of state

```

state-space : master-switch.status = on;
air.temp < controller.tr + 2;
motor.status = on;
motor.speed = adequate;
water.temp >= controller.tw;
fuel-sensor.status = safe;
combustion-sensor.status = safe;
water-valve.status = open;
oil-valve.status = open

```

#### SHUTDOWN instance-of state

```

state-space : master-switch.status = on;
air.temp >= controller.tr + 2;
motor.status = on;
fuel-sensor.status = safe;
combustion-sensor.status = safe;
water-valve.status = open;
oil-valve.status = closed;
five-sec-timer.status = on

```

#### ABNORMAL-SHUTDOWN instance-of state

```

state-space : motor.status = off;
motor.speed = inadequate;
%           water-valve.status = open; % execution showed we didn't need
oil-valve.status = closed;
five-sec-timer.status = on

```

#### WAIT5MINUTES instance-of state

```

state-space : master-switch.status = on;
motor.status = off;
fuel-sensor.status = safe;
combustion-sensor.status = safe;
water-valve.status = closed;
oil-valve.status = closed;
ignition.status = off;
five-sec-timer.status = off;
five-min-timer.status = on

```

#### HOLD instance-of state

```

state-space : five-sec-timer.status = off;
five-min-timer.status = off;
motor.status = off;
water-valve.status = closed;
oil-valve.status = closed;

```

ignition.status = off

%%%% EVENTS %%%%

%% Internal Events

MASTER-SWITCH-ON instance-of event  
type: internal

MOTOR-TURNED-ON instance-of event  
type: internal

OIL-IGNITED instance-of event  
type: internal

WATER-VALVE-OPENED instance-of event  
type: internal

DONE-HEATING-WATER instance-of event  
type: internal

MOTOR-TURNED-OFF instance-of event  
type: internal

DONE-WAITING instance-of event  
type: internal

MASTER-SWITCH-OFF instance-of event  
type: internal

ABNORMAL-FUEL instance-of event  
type: internal

ABNORMAL-COMBUSTION instance-of event  
type: internal

SYSTEM-IS-RESET instance-of event  
type: internal

SYSTEM-IS-OFF instance-of event  
type: internal

%% External Events

SWITCH-TURNED-ON instance-of event  
type: external

AIR-TEMP-BELOW-PRESET instance-of event  
type: external

ADEQUATE-MOTOR-SPEED instance-of event  
     type: external  
  
 WATER-TEMP-ABOVE-PRESET instance-of event  
     type: external  
  
 AIR-TEMP-ABOVE-PRESET instance-of event  
     type: external  
  
 FIVE-SEC-TIMER-EXPIRES instance-of event  
     type: external  
  
 FIVE-MIN-TIMER-EXPIRES instance-of event  
     type: external  
  
 SWITCH-TURNED-OFF instance-of event  
     type: external  
  
 UNSAFE-COMBUSTION-SENSOR instance-of event  
     type: external  
  
 UNSAFE-FUEL-SENSOR instance-of event  
     type: external  
  
 RESET-SYSTEM instance-of event  
     type: external  
  
 SYSTEM-TURNED-OFF instance-of event  
     type: external

%%%%% BEHAVIORS - STATE ACTIVITIES %%%%%

FURNACE-OFF instance-of behavior  
     master-switch.status, = on  
     -->  
     event,                      MASTER-SWITCH-ON

%%%%%

FURNACE-IDLE instance-of behavior

air.temp,                      < controller.tr - 2,              dont-care;  
 master-switch.status,        = on,                              = off  
 -->  
 motor.status,                on,                              off  
 event,                        MOTOR-TURNED-ON,        MASTER-SWITCH-OFF

%%%%%



# FURNACE-MOTOR-ON instance-of behavior

```
motor.speed,          dont-care,          = adequate;
master-switch.status, = off,              = on
-->
ignition.status,      off,                on;
oil-valve.status,     closed,              open;
motor.status,         off,                on
event,               MASTER-SWITCH-OFF,    OIL-IGNITED
```

%%%%

# FURNACE-WATER-HEATING instance-of behavior

```
Water.temp,          > controller.tw,
                    dont-care,  dont-care,  dont-care;
master-switch.status, = on,      = off,      dont-care,  dont-care;
fuel-sensor.status,  = safe,     dont-care,  = unsafe,   dont-care;
combustion-sensor.status, = safe,  dont-care,  dont-care,  = unsafe
-->
water-valve.status,  open,        closed,     closed,     closed;
oil-valve.status,    open,        closed,     closed,     closed;
five-sec-timer.status, off,       on,         on,         on;
motor.status,        on,          off,        off,        off;
motor.speed,         adequate,    inadequate, inadequate, inadequate
event, WATER-VALVE-OPENED, MASTER-SWITCH-OFF, ABNORMAL-FUEL,
                                                ABNORMAL-COMBUSTION
```

%%%%

# FURNACE-RUNNING instance-of behavior

```
air.temp,            >= controller.tr + 2,
                    dont-care,  dont-care,  dont-care;
fuel-sensor.status,  = safe,     = unsafe,   dont-care,  dont-care;
combustion-sensor.status, = safe,  dont-care,  = unsafe,   dont-care;
master-switch.status, = on,       = on,       = on,       = off
-->
oil-valve.status,    closed,     closed,     closed,     closed;
five-sec-timer.status, on,        on,         on,         on;
motor.status,        on,          off,        off,        off;
motor.speed,         adequate,    inadequate, inadequate, inadequate
event, DONE-HEATING-WATER, ABNORMAL-FUEL, ABNORMAL-COMBUSTION,
                                                MASTER-SWITCH-OFF
```

%%%%

# FURNACE-SHUTTING-DOWN instance-of behavior

```
five-sec-timer.status, = off;
fuel-sensor.status,    = safe;
```

```

combustion-sensor.status, = safe
-->
motor.status,             off;
motor.speed,              inadequate;
water-valve.status,      closed;
five-min-timer.status,   on;
ignition.status,         off;
water.temp,              controller.tw - 2
event,                   MOTOR-TURNED-OFF

```

%%%%%

ABNORMAL-FURNACE-SHUTTING-DOWN instance-of behavior

```

five-sec-timer.status,   = off
-->
motor.status,           off;
motor.speed,            inadequate;
water-valve.status,     closed;
five-min-timer.status,  off;
ignition.status,        off;
water.temp,             controller.tw - 2
event,                  MOTOR-TURNED-OFF

```

%%%%%

FURNACE-WAITING instance-of behavior

```

five-min-timer.status,   = off
-->
event,                   DONE-WAITING

```

%%%%%

FURNACE-ABNORMAL instance-of behavior

```

fuel-sensor.status,      = safe,      = safe;
combustion-sensor.status, = safe,      = safe;
master-switch.status,    = off,       = on
-->
event,                   SYSTEM-IS-OFF,  SYSTEM-IS-RESET

```

%%%%%%%%%%%% BEHAVIORS - EVENT ACTIONS %%%%%%%%%%

SWITCH-TURNED-ON-BEH instance-of behavior

```

true
-->
master-switch.status := on
event none

```

AIR-TEMP-BELOW-PRESET-BEH instance-of behavior

```
true
-->
air.temp := controller.tr - 3
event none
```

ADEQUATE-MOTOR-SPEED-BEH instance-of behavior

```
true
-->
motor.speed := adequate
event none
```

WATER-TEMP-ABOVE-PRESET-BEH instance-of behavior

```
true
-->
water.temp := controller.tw + 1
event none
```

AIR-TEMP-ABOVE-PRESET-BEH instance-of behavior

```
true
-->
air.temp := controller.tr + 3
event none
```

FIVE-SEC-TIMER-EXPIRES-BEH instance-of behavior

```
true
-->
five-sec-timer.status := off
event none
```

FIVE-MIN-TIMER-EXPIRES-BEH instance-of behavior

```
true
-->
five-min-timer.status := off
event none
```

SWITCH-TURNED-OFF-BEH instance-of behavior

```
true
-->
master-switch.status := off
event none
```

UNSAFE-COMBUSTION-SENSOR-BEH instance-of behavior

```
true
-->
combustion-sensor.status := unsafe
event none
```

UNSAFE-FUEL-SENSOR-BEH instance-of behavior

```
true
```

```

-->
fuel-sensor.status := unsafe
event none

RESET-SYSTEM-BEH instance-of behavior
true
-->
fuel-sensor.status := safe &
combustion-sensor.status := safe &
master-switch.status := on
event none

SYSTEM-TURNED-OFF-BEH instance-of behavior
true
-->
fuel-sensor.status := safe &
combustion-sensor.status := safe &
master-switch.status := off
event none

```

# %%% Relation Table

TABLE1 instance-of relation-table

%%FROM-OBJECT      ASSOCIATION TO-OBJECT

%% STATE- INTERNAL-EVENT RELATIONSHIPS

OFF,	MASTER-SWITCH-ON,	IDLE;
IDLE,	MOTOR-TURNED-ON,	MOTOR-ON;
MOTOR-ON,	OIL-IGNITED,	WATER-HEATING;
WATER-HEATING,	WATER-VALVE-OPENED,	RUNNING;
RUNNING,	DONE-HEATING-WATER,	SHUTDOWN;
SHUTDOWN,	MOTOR-TURNED-OFF,	WAIT5MINUTES;
ABNORMAL-SHUTDOWN,	MOTOR-TURNED-OFF,	HOLD;
WAIT5MINUTES,	DONE-WAITING,	IDLE;
IDLE,	MASTER-SWITCH-OFF,	OFF;
MOTOR-ON,	MASTER-SWITCH-OFF,	OFF;
WATER-HEATING,	MASTER-SWITCH-OFF,	ABNORMAL-SHUTDOWN;
WATER-HEATING,	ABNORMAL-FUEL,	ABNORMAL-SHUTDOWN;
WATER-HEATING,	ABNORMAL-COMBUSTION,	ABNORMAL-SHUTDOWN;
RUNNING,	MASTER-SWITCH-OFF,	ABNORMAL-SHUTDOWN;
RUNNING,	ABNORMAL-FUEL,	ABNORMAL-SHUTDOWN;
RUNNING,	ABNORMAL-COMBUSTION,	ABNORMAL-SHUTDOWN;
HOLD,	SYSTEM-IS-RESET,	IDLE;
HOLD,	SYSTEM-IS-OFF,	OFF;

%% STATE- EXTERNAL-EVENT RELATIONSHIPS

OUTSIDE,	SWITCH-TURNED-ON,	OFF;
----------	-------------------	------

OUTSIDE,	AIR-TEMP-BELOW-PRESET,	IDLE;
OUTSIDE,	ADEQUATE-MOTOR-SPEED,	MOTOR-ON;
OUTSIDE,	WATER-TEMP-ABOVE-PRESET,	WATER-HEATING;
OUTSIDE,	AIR-TEMP-ABOVE-PRESET,	RUNNING;
OUTSIDE,	FIVE-SEC-TIMER-EXPIRES,	SHUTDOWN;
OUTSIDE,	FIVE-SEC-TIMER-EXPIRES,	ABNORMAL-SHUTDOWN;
OUTSIDE,	FIVE-MIN-TIMER-EXPIRES,	WAIT5MINUTES;
OUTSIDE,	SWITCH-TURNED-OFF,	IDLE;
OUTSIDE,	SWITCH-TURNED-OFF,	MOTOR-ON;
OUTSIDE,	SWITCH-TURNED-OFF,	WATER-HEATING;
OUTSIDE,	SWITCH-TURNED-OFF,	RUNNING;
OUTSIDE,	UNSAFE-COMBUSTION-SENSOR,	WATER-HEATING;
OUTSIDE,	UNSAFE-COMBUSTION-SENSOR,	RUNNING;
OUTSIDE,	UNSAFE-FUEL-SENSOR,	WATER-HEATING;
OUTSIDE,	UNSAFE-FUEL-SENSOR,	RUNNING;
OUTSIDE,	RESET-SYSTEM,	HOLD;
OUTSIDE,	SYSTEM-TURNED-OFF,	HOLD;

#### %%Event-Behavior-relationships

SWITCH-TURNED-ON,	ICO,	SWITCH-TURNED-ON-BEH;
AIR-TEMP-BELOW-PRESET,	ICO,	AIR-TEMP-BELOW-PRESET-BEH;
ADEQUATE-MOTOR-SPEED,	ICO,	ADEQUATE-MOTOR-SPEED-BEH;
WATER-TEMP-ABOVE-PRESET,	ICO,	WATER-TEMP-ABOVE-PRESET-BEH;
AIR-TEMP-ABOVE-PRESET,	ICO,	AIR-TEMP-ABOVE-PRESET-BEH;
FIVE-SEC-TIMER-EXPIRES,	ICO,	FIVE-SEC-TIMER-EXPIRES-BEH;
FIVE-MIN-TIMER-EXPIRES,	ICO,	FIVE-MIN-TIMER-EXPIRES-BEH;
SWITCH-TURNED-OFF,	ICO,	SWITCH-TURNED-OFF-BEH;
UNSAFE-COMBUSTION-SENSOR,	ICO,	UNSAFE-COMBUSTION-SENSOR-BEH;
UNSAFE-FUEL-SENSOR,	ICO,	UNSAFE-FUEL-SENSOR-BEH;
RESET-SYSTEM,	ICO,	RESET-SYSTEM-BEH;
SYSTEM-TURNED-OFF,	ICO,	SYSTEM-TURNED-OFF-BEH;

#### %% ENTITY-RELATIONSHIPS

CONTROLLER,	ACTIVATES,	MOTOR;
CONTROLLER,	MONITORS,	MOTOR;
CONTROLLER,	MONITORS,	THERMOSTAT;
CONTROLLER,	POSITIONS,	VALVE;
CONTROLLER,	MONITORS,	MASTER-SWITCH;
CONTROLLER,	MONITORS,	SENSOR;
CONTROLLER,	SETS,	TIMER;

#### %% STATE-BEHAVIOR RELATIONSHIPS

OFF,	ICO,	FURNACE-OFF ;
IDLE,	ICO,	FURNACE-IDLE;
MOTOR-ON,	ICO,	FURNACE-MOTOR-ON;
WATER-HEATING,	ICO,	FURNACE-WATER-HEATING;
RUNNING,	ICO,	FURNACE-RUNNING;
SHUTDOWN,	ICO,	FURNACE-SHUTTING-DOWN;

ABNORMAL-SHUTDOWN, ICO,  
WAIT5MINUTES, ICO,  
HOLD, ICO,

ABNORMAL-FURNACE-SHUTTING-DOWN;  
FURNACE-WAITING;  
FURNACE-ABNORMAL.

#### *D.6 Heater Problem REFINE Executable Specification*

```
!! in-package ('RU)
!! in-grammar ('user)

var OML-Obj : object-class subtype-of user-object

var HOME-HEATER : object-class subtype-of OML-Obj

type return-values = tuple(
    validity: symbol,
    events: seq(symbol),
    behaviors : seq(symbol),
    st-behaviors : seq(symbol))

%%% Define object classes

var THERMOSTAT : object-class subtype-of HOME-HEATER
var THERMOSTAT-TEMP: map(THERMOSTAT, integer) = {}

var TIMER : object-class subtype-of HOME-HEATER
var TIMER-STATUS: map(TIMER, symbol) = {}

var VALVE : object-class subtype-of HOME-HEATER
var VALVE-STATUS: map(VALVE, symbol) = {}

var SENSOR : object-class subtype-of HOME-HEATER
var SENSOR-STATUS: map(SENSOR, symbol) = {}

%%% Define instances of object classes

var CONTROLLER-ENTITY : object-class subtype-of HOME-HEATER
var CONTROLLER-ENTITY-TR: map(CONTROLLER-ENTITY, integer) = {}
var CONTROLLER-ENTITY-TW: map(CONTROLLER-ENTITY, integer) = {}

var CONTROLLER : CONTROLLER-ENTITY =
    set-attrs(make-object('CONTROLLER-ENTITY),
        'name, '*CONTROLLER,
        'CONTROLLER-ENTITY-TR, 70,
        'CONTROLLER-ENTITY-TW, 180)

var IGNITION-ENTITY : object-class subtype-of HOME-HEATER
var IGNITION-ENTITY-STATUS: map(IGNITION-ENTITY, symbol) = {}

var IGNITION : IGNITION-ENTITY =
    set-attrs(make-object('IGNITION-ENTITY),
        'name, '*IGNITION,
        'IGNITION-ENTITY-STATUS, 'OFF)

var MOTOR-ENTITY : object-class subtype-of HOME-HEATER
var MOTOR-ENTITY-STATUS: map(MOTOR-ENTITY, symbol) = {}
var MOTOR-ENTITY-SPEED: map(MOTOR-ENTITY, symbol) = {}

var MOTOR : MOTOR-ENTITY =
    set-attrs(make-object('MOTOR-ENTITY),
        'name, '*MOTOR,
        'MOTOR-ENTITY-STATUS, 'OFF,
```

```

        'MOTOR-ENTITY-SPEED, 'INADEQUATE)

var MASTER-SWITCH-ENTITY : object-class subtype-of HOME-HEATER
var MASTER-SWITCH-ENTITY-STATUS: map(MASTER-SWITCH-ENTITY, symbol) = {}

var MASTER-SWITCH : MASTER-SWITCH-ENTITY =
    set-attrs(make-object('MASTER-SWITCH-ENTITY),
        'name, '*MASTER-SWITCH,
        'MASTER-SWITCH-ENTITY-STATUS, 'OFF)

var WATER : THERMOSTAT =
    set-attrs(make-object('THERMOSTAT),
        'name, '*WATER,
        'THERMOSTAT-TEMP, 60)

var AIR : THERMOSTAT =
    set-attrs(make-object('THERMOSTAT),
        'name, '*AIR,
        'THERMOSTAT-TEMP, 60)

var FIVE-SEC-TIMER : TIMER =
    set-attrs(make-object('TIMER),
        'name, '*FIVE-SEC-TIMER,
        'TIMER-STATUS, 'OFF)

var FIVE-MIN-TIMER : TIMER =
    set-attrs(make-object('TIMER),
        'name, '*FIVE-MIN-TIMER,
        'TIMER-STATUS, 'OFF)

var OIL-VALVE : VALVE =
    set-attrs(make-object('VALVE),
        'name, '*OIL-VALVE,
        'VALVE-STATUS, 'CLOSED)

var WATER-VALVE : VALVE =
    set-attrs(make-object('VALVE),
        'name, '*WATER-VALVE,
        'VALVE-STATUS, 'CLOSED)

var COMBUSTION-SENSOR : SENSOR =
    set-attrs(make-object('SENSOR),
        'name, '*COMBUSTION-SENSOR,
        'SENSOR-STATUS, 'SAFE)

var FUEL-SENSOR : SENSOR =
    set-attrs(make-object('SENSOR),
        'name, '*FUEL-SENSOR,
        'SENSOR-STATUS, 'SAFE)

%%% Define Store Objects

%%% Define objects for each flow object

%%% Define functions for behavior objects

```



```

function SYSTEM-TURNED-OFF-BEH() =
  let(return-symbol : symbol = undefined)

  ( (if true
    then
      (SENSOR-STATUS(FUEL-SENSOR) <- 'SAFE);
      (SENSOR-STATUS(COMBUSTION-SENSOR) <- 'SAFE);
      (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) <- 'OFF)
    ));
  return-symbol

function RESET-SYSTEM-BEH() =
  let(return-symbol : symbol = undefined)

  ( (if true
    then
      (SENSOR-STATUS(FUEL-SENSOR) <- 'SAFE);
      (SENSOR-STATUS(COMBUSTION-SENSOR) <- 'SAFE);
      (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) <- 'ON)
    ));
  return-symbol

function UNSAFE-FUEL-SENSOR-BEH() =
  let(return-symbol : symbol = undefined)

  ( (if true
    then
      (SENSOR-STATUS(FUEL-SENSOR) <- 'UNSAFE)
    ));
  return-symbol

function UNSAFE-COMBUSTION-SENSOR-BEH() =
  let(return-symbol : symbol = undefined)

  ( (if true
    then
      (SENSOR-STATUS(COMBUSTION-SENSOR) <- 'UNSAFE)
    ));
  return-symbol

function SWITCH-TURNED-OFF-BEH() =
  let(return-symbol : symbol = undefined)

  ( (if true
    then
      (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) <- 'OFF)
    ));
  return-symbol

function FIVE-MIN-TIMER-EXPIRES-BEH() =
  let(return-symbol : symbol = undefined)

  ( (if true
    then
      (TIMER-STATUS(FIVE-MIN-TIMER) <- 'OFF)
    ));

```

```

    return-symbol

function FIVE-SEC-TIMER-EXPIRES-BEH() =
    let(return-symbol : symbol = undefined)

    ( (if true
        then
        (TIMER-STATUS(FIVE-SEC-TIMER) <- 'OFF)
    ));
    return-symbol

function AIR-TEMP-ABOVE-PRESET-BEH() =
    let(return-symbol : symbol = undefined)

    ( (if true
        then
        (THERMOSTAT-TEMP(AIR) <- (CONTROLLER-ENTITY-TR(CONTROLLER) + 3))
    ));
    return-symbol

function WATER-TEMP-ABOVE-PRESET-BEH() =
    let(return-symbol : symbol = undefined)

    ( (if true
        then
        (THERMOSTAT-TEMP(WATER) <- (CONTROLLER-ENTITY-TW(CONTROLLER) + 1))
    ));
    return-symbol

function ADEQUATE-MOTOR-SPEED-BEH() =
    let(return-symbol : symbol = undefined)

    ( (if true
        then
        (MOTOR-ENTITY-SPEED(MOTOR) <- 'ADEQUATE)
    ));
    return-symbol

function AIR-TEMP-BELOW-PRESET-BEH() =
    let(return-symbol : symbol = undefined)

    ( (if true
        then
        (THERMOSTAT-TEMP(AIR) <- (CONTROLLER-ENTITY-TR(CONTROLLER) - 3))
    ));
    return-symbol

function SWITCH-TURNED-ON-BEH() =
    let(return-symbol : symbol = undefined)

    ( (if true
        then
        (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) <- 'ON)
    ));
    return-symbol

```

```

function FURNACE-ABNORMAL() : symbol =
  let(return-symbol : symbol = undefined)

(
  (SENSOR-STATUS(FUEL-SENSOR) = 'SAFE)
  & (SENSOR-STATUS(COMBUSTION-SENSOR) = 'SAFE)
  & (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'OFF)
  -->
  (return-symbol <- 'OFF);

  (SENSOR-STATUS(FUEL-SENSOR) = 'SAFE)
  & (SENSOR-STATUS(COMBUSTION-SENSOR) = 'SAFE)
  & (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON)
  -->
  (return-symbol <- 'IDLE)

);
return-symbol
%%%%%%%%%%%%%%

function FURNACE-WAITING() : symbol =
  let(return-symbol : symbol = undefined)

(
  (TIMER-STATUS(FIVE-MIN-TIMER) = 'OFF)
  -->
  (return-symbol <- 'IDLE)

);
return-symbol
%%%%%%%%%%%%%%

function ABNORMAL-FURNACE-SHUTTING-DOWN() : symbol =
  let(return-symbol : symbol = undefined)

(
  (TIMER-STATUS(FIVE-SEC-TIMER) = 'OFF)
  -->
  (MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
  & (MOTOR-ENTITY-SPEED(MOTOR) <- 'INADEQUATE)
  & (VALVE-STATUS(WATER-VALVE) <- 'CLOSED)
  & (TIMER-STATUS(FIVE-MIN-TIMER) <- 'OFF)
  & (IGNITION-ENTITY-STATUS(IGNITION) <- 'OFF)
  & (THERMOSTAT-TEMP(WATER) <- (CONTROLLER-ENTITY-TW(CONTROLLER) - 2))
  & (return-symbol <- 'HOLD)

);
return-symbol
%%%%%%%%%%%%%%

function FURNACE-SHUTTING-DOWN() : symbol =
  let(return-symbol : symbol = undefined)

(
  (TIMER-STATUS(FIVE-SEC-TIMER) = 'OFF)
  & (SENSOR-STATUS(FUEL-SENSOR) = 'SAFE)

```

```

& (SENSOR-STATUS(COMBUSTION-SENSOR) = 'SAFE)
-->
(MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
& (MOTOR-ENTITY-SPEED(MOTOR) <- 'INADEQUATE)
& (VALVE-STATUS(WATER-VALVE) <- 'CLOSED)
& (TIMER-STATUS(FIVE-MIN-TIMER) <- 'ON)
& (IGNITION-ENTITY-STATUS(IGNITION) <- 'OFF)
& (THERMOSTAT-TEMP(WATER) <- (CONTROLLER-ENTITY-TW(CONTROLLER) - 2))
& (return-symbol <- 'WAIT5MINUTES)

);
return-symbol
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function FURNACE-RUNNING() : symbol =
  let(return-symbol : symbol = undefined)

(
  (THERMOSTAT-TEMP(AIR) >= (CONTROLLER-ENTITY-TR(CONTROLLER) + 2))
  & (SENSOR-STATUS(FUEL-SENSOR) = 'SAFE)
  & (SENSOR-STATUS(COMBUSTION-SENSOR) = 'SAFE)
  & (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON)
  -->
  (VALVE-STATUS(OIL-VALVE) <- 'CLOSED)
  & (TIMER-STATUS(FIVE-SEC-TIMER) <- 'ON)
  & (MOTOR-ENTITY-STATUS(MOTOR) <- 'ON)
  & (MOTOR-ENTITY-SPEED(MOTOR) <- 'ADEQUATE)
  & (return-symbol <- 'SHUTDOWN);

  (SENSOR-STATUS(FUEL-SENSOR) = 'UNSAFE)
  & (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON)
  -->
  (VALVE-STATUS(OIL-VALVE) <- 'CLOSED)
  & (TIMER-STATUS(FIVE-SEC-TIMER) <- 'ON)
  & (MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
  & (MOTOR-ENTITY-SPEED(MOTOR) <- 'INADEQUATE)
  & (return-symbol <- 'ABNORMAL-SHUTDOWN);

  (SENSOR-STATUS(COMBUSTION-SENSOR) = 'UNSAFE)
  & (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON)
  -->
  (VALVE-STATUS(OIL-VALVE) <- 'CLOSED)
  & (TIMER-STATUS(FIVE-SEC-TIMER) <- 'ON)
  & (MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
  & (MOTOR-ENTITY-SPEED(MOTOR) <- 'INADEQUATE)
  & (return-symbol <- 'ABNORMAL-SHUTDOWN);

  (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'OFF)
  -->
  (VALVE-STATUS(OIL-VALVE) <- 'CLOSED)
  & (TIMER-STATUS(FIVE-SEC-TIMER) <- 'ON)
  & (MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
  & (MOTOR-ENTITY-SPEED(MOTOR) <- 'INADEQUATE)
  & (return-symbol <- 'ABNORMAL-SHUTDOWN)

);

```

```

return-symbol
XXXXXXXXXXXX

function FURNACE-WATER-HEATING() : symbol =
  let(return-symbol : symbol = undefined)

  (
    (THERMOSTAT-TEMP(WATER) > CONTROLLER-ENTITY-TW(CONTROLLER))
    & (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON)
    & (SENSOR-STATUS(FUEL-SENSOR) = 'SAFE)
    & (SENSOR-STATUS(COMBUSTION-SENSOR) = 'SAFE)
    -->
    (VALVE-STATUS(WATER-VALVE) <- 'OPEN)
    & (VALVE-STATUS(OIL-VALVE) <- 'OPEN)
    & (TIMER-STATUS(FIVE-SEC-TIMER) <- 'OFF)
    & (MOTOR-ENTITY-STATUS(MOTOR) <- 'ON)
    & (MOTOR-ENTITY-SPEED(MOTOR) <- 'ADEQUATE)
    & (return-symbol <- 'RUNNING);

    (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'OFF)
    -->
    (VALVE-STATUS(WATER-VALVE) <- 'CLOSED)
    & (VALVE-STATUS(OIL-VALVE) <- 'CLOSED)
    & (TIMER-STATUS(FIVE-SEC-TIMER) <- 'ON)
    & (MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
    & (MOTOR-ENTITY-SPEED(MOTOR) <- 'INADEQUATE)
    & (return-symbol <- 'ABNORMAL-SHUTDOWN);

    (SENSOR-STATUS(FUEL-SENSOR) = 'UNSAFE)
    -->
    (VALVE-STATUS(WATER-VALVE) <- 'CLOSED)
    & (VALVE-STATUS(OIL-VALVE) <- 'CLOSED)
    & (TIMER-STATUS(FIVE-SEC-TIMER) <- 'ON)
    & (MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
    & (MOTOR-ENTITY-SPEED(MOTOR) <- 'INADEQUATE)
    & (return-symbol <- 'ABNORMAL-SHUTDOWN);

    (SENSOR-STATUS(COMBUSTION-SENSOR) = 'UNSAFE)
    -->
    (VALVE-STATUS(WATER-VALVE) <- 'CLOSED)
    & (VALVE-STATUS(OIL-VALVE) <- 'CLOSED)
    & (TIMER-STATUS(FIVE-SEC-TIMER) <- 'ON)
    & (MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
    & (MOTOR-ENTITY-SPEED(MOTOR) <- 'INADEQUATE)
    & (return-symbol <- 'ABNORMAL-SHUTDOWN)

  );
return-symbol
XXXXXXXXXXXX

function FURNACE-MOTOR-ON() : symbol =
  let(return-symbol : symbol = undefined)

  (
    (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'OFF)
    -->

```

```

(IGNITION-ENTITY-STATUS(IGNITION) <- 'OFF)
& (VALVE-STATUS(OIL-VALVE) <- 'CLOSED)
& (MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
& (return-symbol <- 'OFF);

(MOTOR-ENTITY-SPEED(MOTOR) = 'ADEQUATE)
& (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON)
-->
(IGNITION-ENTITY-STATUS(IGNITION) <- 'ON)
& (VALVE-STATUS(OIL-VALVE) <- 'OPEN)
& (MOTOR-ENTITY-STATUS(MOTOR) <- 'ON)
& (return-symbol <- 'WATER-HEATING)

);
return-symbol
XXXXXXXXXXXX

function FURNACE-IDLE() : symbol =
  let(return-symbol : symbol = undefined)

(
  (THERMOSTAT-TEMP(AIR) < (CONTROLLER-ENTITY-TR(CONTROLLER) - 2))
  & (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON)
  -->
  (MOTOR-ENTITY-STATUS(MOTOR) <- 'ON)
  & (return-symbol <- 'MOTOR-ON);

  (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'OFF)
  -->
  (MOTOR-ENTITY-STATUS(MOTOR) <- 'OFF)
  & (return-symbol <- 'OFF)

);
return-symbol
XXXXXXXXXXXX

function FURNACE-OFF() : symbol =
  let(return-symbol : symbol = undefined)

(
  (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON)
  -->
  (return-symbol <- 'IDLE)

);
return-symbol
XXXXXXXXXXXX

%%% Define function for each state object

function HOLD() : return-values =
  let (valid-ext-event : seq(symbol) =
    ['SYSTEM-TURNED-OFF, 'RESET-SYSTEM],
    valid-event-beh : seq(symbol) =
    ['SYSTEM-TURNED-OFF-BEH, 'RESET-SYSTEM-BEH],
    state-beh : seq(symbol) = ['FURNACE-ABNORMAL],

```

```

    return-tuple : return-values = undefined)

format(true, "The current state of the system is HOLD");
(if
  (TIMER-STATUS(FIVE-SEC-TIMER) = 'OFF) and
  (TIMER-STATUS(FIVE-MIN-TIMER) = 'OFF) and
  (MOTOR-ENTITY-STATUS(MOTOR) = 'OFF) and
  (VALVE-STATUS(WATER-VALVE) = 'CLOSED) and
  (VALVE-STATUS(OIL-VALVE) = 'CLOSED) and
  (IGNITION-ENTITY-STATUS(IGNITION) = 'OFF)
  then
    format(true, "~%  VALID STATE SPACE~%");
    return-tuple <- <'valid,  valid-ext-event, valid-event-beh, state-beh>
  else
    format(true, "~%  INVALID STATE SPACE~%");
    return-tuple <- <'invalid,  [], ['HOLD], state-beh>);
return-tuple

function WAIT5MINUTES() : return-values =
let (valid-ext-event : seq(symbol) =
    ['FIVE-MIN-TIMER-EXPIRES],
    valid-event-beh : seq(symbol) =
    ['FIVE-MIN-TIMER-EXPIRES-BEH],
    state-beh : seq(symbol) = ['FURNACE-WAITING],
    return-tuple : return-values = undefined)

format(true, "The current state of the system is WAIT5MINUTES");
(if
  (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON) and
  (MOTOR-ENTITY-STATUS(MOTOR) = 'OFF) and
  (SENSOR-STATUS(FUEL-SENSOR) = 'SAFE) and
  (SENSOR-STATUS(COMBUSTION-SENSOR) = 'SAFE) and
  (VALVE-STATUS(WATER-VALVE) = 'CLOSED) and
  (VALVE-STATUS(OIL-VALVE) = 'CLOSED) and
  (IGNITION-ENTITY-STATUS(IGNITION) = 'OFF) and
  (TIMER-STATUS(FIVE-SEC-TIMER) = 'OFF) and
  (TIMER-STATUS(FIVE-MIN-TIMER) = 'ON)
  then
    format(true, "~%  VALID STATE SPACE~%");
    return-tuple <- <'valid,  valid-ext-event, valid-event-beh, state-beh>
  else
    format(true, "~%  INVALID STATE SPACE~%");
    return-tuple <- <'invalid,  [], ['WAIT5MINUTES], state-beh>);
return-tuple

function ABNORMAL-SHUTDOWN() : return-values =
let (valid-ext-event : seq(symbol) =
    ['FIVE-SEC-TIMER-EXPIRES],
    valid-event-beh : seq(symbol) =
    ['FIVE-SEC-TIMER-EXPIRES-BEH],
    state-beh : seq(symbol) = ['ABNORMAL-FURNACE-SHUTTING-DOWN],
    return-tuple : return-values = undefined)

format(true, "The current state of the system is ABNORMAL-SHUTDOWN");
(if
  (MOTOR-ENTITY-STATUS(MOTOR) = 'OFF) and

```

```

(MOTOR-ENTITY-SPEED(MOTOR) = 'INADEQUATE) and
(VALVE-STATUS(OIL-VALVE) = 'CLOSED) and
(TIMER-STATUS(FIVE-SEC-TIMER) = 'ON)
then
  format(true, "~%  VALID STATE SPACE~%");
  return-tuple <- <'valid,  valid-ext-event, valid-event-beh, state-beh>
else
  format(true, "~%  INVALID STATE SPACE~%");
  return-tuple <- <'invalid,  [], ['ABNORMAL-SHUTDOWN], state-beh>);
return-tuple

function SHUTDOWN() : return-values =
let (valid-ext-event : seq(symbol) =
    ['FIVE-SEC-TIMER-EXPIRES],
    valid-event-beh : seq(symbol) =
    ['FIVE-SEC-TIMER-EXPIRES-BEH],
    state-beh : seq(symbol) = ['FURNACE-SHUTTING-DOWN],
    return-tuple : return-values = undefined)

format(true, "The current state of the system is SHUTDOWN");
(if
  (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON) and
  (THERMOSTAT-TEMP(AIR) >= (CONTROLLER-ENTITY-TR(CONTROLLER) + 2)) and
  (MOTOR-ENTITY-STATUS(MOTOR) = 'ON) and
  (SENSOR-STATUS(FUEL-SENSOR) = 'SAFE) and
  (SENSOR-STATUS(COMBUSTION-SENSOR) = 'SAFE) and
  (VALVE-STATUS(WATER-VALVE) = 'OPEN) and
  (VALVE-STATUS(OIL-VALVE) = 'CLOSED) and
  (TIMER-STATUS(FIVE-SEC-TIMER) = 'ON)
  then
    format(true, "~%  VALID STATE SPACE~%");
    return-tuple <- <'valid,  valid-ext-event, valid-event-beh, state-beh>
  else
    format(true, "~%  INVALID STATE SPACE~%");
    return-tuple <- <'invalid,  [], ['SHUTDOWN], state-beh>);
return-tuple

function RUNNING() : return-values =
let (valid-ext-event : seq(symbol) =
    ['UNSAFE-FUEL-SENSOR, 'UNSAFE-COMBUSTION-SENSOR,
    'SWITCH-TURNED-OFF, 'AIR-TEMP-ABOVE-PRESET],
    valid-event-beh : seq(symbol) =
    ['UNSAFE-FUEL-SENSOR-BEH, 'UNSAFE-COMBUSTION-SENSOR-BEH,
    'SWITCH-TURNED-OFF-BEH, 'AIR-TEMP-ABOVE-PRESET-BEH],
    state-beh : seq(symbol) = ['FURNACE-RUNNING],
    return-tuple : return-values = undefined)

format(true, "The current state of the system is RUNNING");
(if
  (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON) and
  (THERMOSTAT-TEMP(AIR) < (CONTROLLER-ENTITY-TR(CONTROLLER) + 2)) and
  (MOTOR-ENTITY-STATUS(MOTOR) = 'ON) and
  (MOTOR-ENTITY-SPEED(MOTOR) = 'ADEQUATE) and
  (THERMOSTAT-TEMP(WATER) >= CONTROLLER-ENTITY-TW(CONTROLLER)) and
  (SENSOR-STATUS(FUEL-SENSOR) = 'SAFE) and
  (SENSOR-STATUS(COMBUSTION-SENSOR) = 'SAFE) and

```



```

        (VALVE-STATUS(WATER-VALVE) = 'OPEN) and
        (VALVE-STATUS(OIL-VALVE) = 'OPEN)
    then
        format(true, "~%  VALID STATE SPACE~%");
        return-tuple <- <'valid, valid-ext-event, valid-event-beh, state-beh>
    else
        format(true, "~%  INVALID STATE SPACE~%");
        return-tuple <- <'invalid, [], ['RUNNING], state-beh>);
    return-tuple

function WATER-HEATING() : return-values =
let (valid-ext-event : seq(symbol) =
    ['UNSAFE-FUEL-SENSOR, 'UNSAFE-COMBUSTION-SENSOR,
     'SWITCH-TURNED-OFF, 'WATER-TEMP-ABOVE-PRESET],
    valid-event-beh : seq(symbol) =
    ['UNSAFE-FUEL-SENSOR-BEH, 'UNSAFE-COMBUSTION-SENSOR-BEH,
     'SWITCH-TURNED-OFF-BEH, 'WATER-TEMP-ABOVE-PRESET-BEH],
    state-beh : seq(symbol) = ['FURNACE-WATER-HEATING],
    return-tuple : return-values = undefined)

format(true, "The current state of the system is WATER-HEATING");
(if
    (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON) and
    (THERMOSTAT-TEMP(AIR) < (CONTROLLER-ENTITY-TR(CONTROLLER) + 2)) and
    (MOTOR-ENTITY-STATUS(MOTOR) = 'ON) and
    (MOTOR-ENTITY-SPEED(MOTOR) = 'ADEQUATE) and
    (THERMOSTAT-TEMP(WATER) < CONTROLLER-ENTITY-TW(CONTROLLER)) and
    (SENSOR-STATUS(FUEL-SENSOR) = 'SAFE) and
    (SENSOR-STATUS(COMBUSTION-SENSOR) = 'SAFE) and
    (VALVE-STATUS(WATER-VALVE) = 'CLOSED) and
    (VALVE-STATUS(OIL-VALVE) = 'OPEN)
    then
        format(true, "~%  VALID STATE SPACE~%");
        return-tuple <- <'valid, valid-ext-event, valid-event-beh, state-beh>
    else
        format(true, "~%  INVALID STATE SPACE~%");
        return-tuple <- <'invalid, [], ['WATER-HEATING], state-beh>);
    return-tuple

function MOTOR-ON() : return-values =
let (valid-ext-event : seq(symbol) =
    ['SWITCH-TURNED-OFF, 'ADEQUATE-MOTOR-SPEED],
    valid-event-beh : seq(symbol) =
    ['SWITCH-TURNED-OFF-BEH, 'ADEQUATE-MOTOR-SPEED-BEH],
    state-beh : seq(symbol) = ['FURNACE-MOTOR-ON],
    return-tuple : return-values = undefined)

format(true, "The current state of the system is MOTOR-ON");
(if
    (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON) and
    (MOTOR-ENTITY-STATUS(MOTOR) = 'ON) and
    (MOTOR-ENTITY-SPEED(MOTOR) = 'INADEQUATE) and
    (THERMOSTAT-TEMP(AIR) < (CONTROLLER-ENTITY-TR(CONTROLLER) - 2)) and
    (IGNITION-ENTITY-STATUS(IGNITION) = 'OFF) and
    (VALVE-STATUS(OIL-VALVE) = 'CLOSED)
    then

```

```

    format(true, "~%  VALID STATE SPACE~%");
    return-tuple <- <'valid,  valid-ext-event, valid-event-beh, state-beh>
else
    format(true, "~%  INVALID STATE SPACE~%");
    return-tuple <- <'invalid,  [], ['MOTOR-ON], state-beh>);
return-tuple

function IDLE() : return-values =
let (valid-ext-event : seq(symbol) =
    ['SWITCH-TURNED-OFF, 'AIR-TEMP-BELOW-PRESET],
    valid-event-beh : seq(symbol) =
    ['SWITCH-TURNED-OFF-BEH, 'AIR-TEMP-BELOW-PRESET-BEH],
    state-beh : seq(symbol) = ['FURNACE-IDLE],
    return-tuple : return-values = undefined)

format(true, "The current state of the system is IDLE");
(if
    (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'ON) and
    (TIMER-STATUS(FIVE-MIN-TIMER) = 'OFF) and
    (TIMER-STATUS(FIVE-SEC-TIMER) = 'OFF)
then
    format(true, "~%  VALID STATE SPACE~%");
    return-tuple <- <'valid,  valid-ext-event, valid-event-beh, state-beh>
else
    format(true, "~%  INVALID STATE SPACE~%");
    return-tuple <- <'invalid,  [], ['IDLE], state-beh>);
return-tuple

function OFF() : return-values =
let (valid-ext-event : seq(symbol) =
    ['SWITCH-TURNED-ON],
    valid-event-beh : seq(symbol) =
    ['SWITCH-TURNED-ON-BEH],
    state-beh : seq(symbol) = ['FURNACE-OFF],
    return-tuple : return-values = undefined)

format(true, "The current state of the system is OFF");
(if
    (MASTER-SWITCH-ENTITY-STATUS(MASTER-SWITCH) = 'OFF)
then
    format(true, "~%  VALID STATE SPACE~%");
    return-tuple <- <'valid,  valid-ext-event, valid-event-beh, state-beh>
else
    format(true, "~%  INVALID STATE SPACE~%");
    return-tuple <- <'invalid,  [], ['OFF], state-beh>);
return-tuple

function sim() =
let (sfunction : return-values = undefined,
    st-name : symbol = 'OFF,  %% assume first state in OML file is initial
    done : boolean = false,
    reply : integer = undefined)

while ~done do
    sfunction <- funcall(st-name);
    (if sfunction.validity = 'valid then

```

```

reply <- Make-Menu(sfunction.events, "Events that can occur:");
(if Reply <= size(sfunction.events) then
  funcall(sfunction.behaviors(reply));

  enumerate st-beh over sfunction.st-behaviors do
    st-name <- funcall(st-beh)
  elseif Reply = size(sfunction.events)+2 then
    done <- true          %% selects quit
  )
else
  %% not valid state
  done <- true;
format (true, "The system's current state space conflicts with
the state space required to be in the above mentioned state. Here are the
current attribute values in the system. Compare them with the required values
specified in your specification to find the inconsistencies.~%");

  (enumerate obj over [obj | (obj : HOME-HEATER) HOME-HEATER(obj)] do
    (enumerate attr over Return-Attribute-List(obj) do
      format(true, "  A.~A : ~A~%",name(obj), name(attr), retrieve-attribute(obj, attr))
    )))

%%% Define function for each process object

%%% Defines function for erasing all objects in Refine's database.
%%% Execute this function before you reload this file if you do not use
%%% the convert process.

function clear-objects() =
  (enumerate obj over [obj | (obj : HOME-HEATER) HOME-HEATER(obj)] do
    erase-object(obj))

```

## *Appendix E. Library Problem Analysis*

This problem is from the problem set for the Fourth International Workshop on Software Specification and Design. It is based on R.A. Kemmerer's "Testing formal specifications to detect design errors". The initial ERM and DFMs were composed by Blankenship (6:Appendix F), but were modified to improve their understandability.

### *E.1 Problem Statement*

"Consider a small library database with the following transactions:

1. Check out a copy of a book / Return a copy of a book;
2. Add a copy of a book to / Remove a copy of a book from the library;
3. Get the list of books by a particular author or in a particular subject area;
4. Find out the list of books currently checked out by a particular borrower;
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4 and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

1. All copies in the library must be available for checkout or be checked out.
2. No copy of the book may be both available and checked out at the same time.
3. A borrower may not have more than a predefined number of books checked out at one time.

(18)"

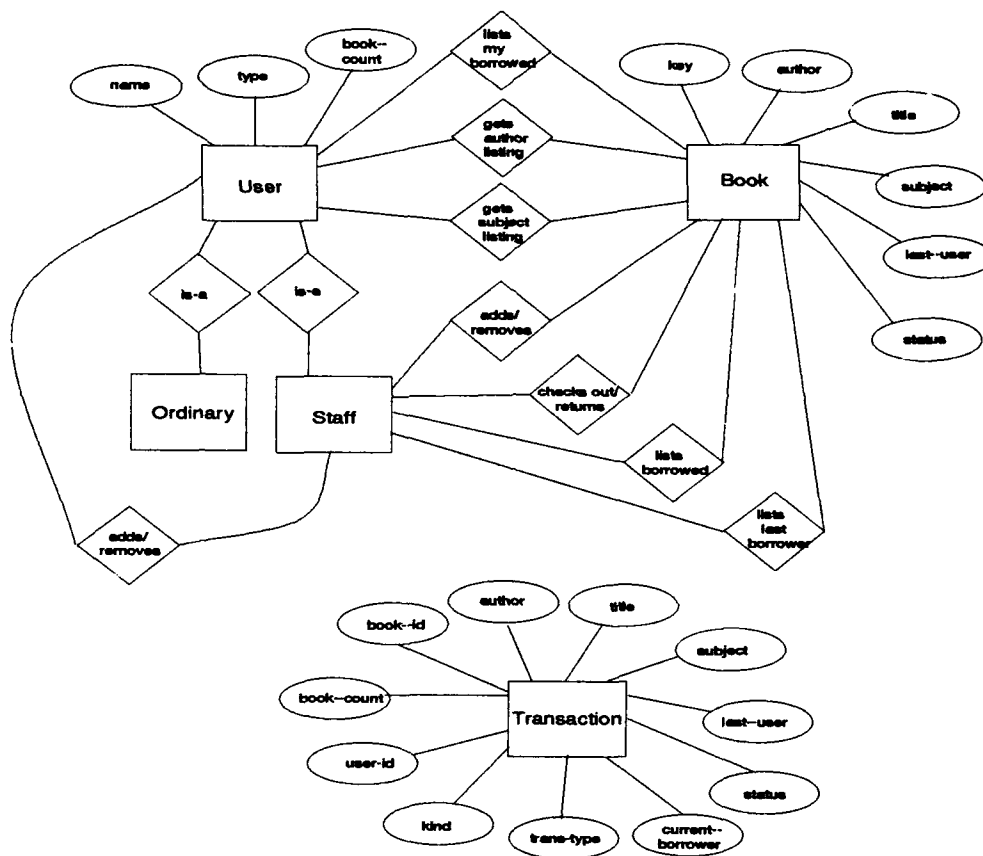


Figure 25. Library: Entity Relationship Model (6:F-12)

## E.2 Entity-Relationship Models

The entity relationship model shows the primary objects in the system and the relationships between them. The library system contains books and users. The relationships in the diagram identify operations that will be required on users and books. A transaction is an entity that is used to transport information from the user interface to the process needing that information.

## E.3 Data Flow Models

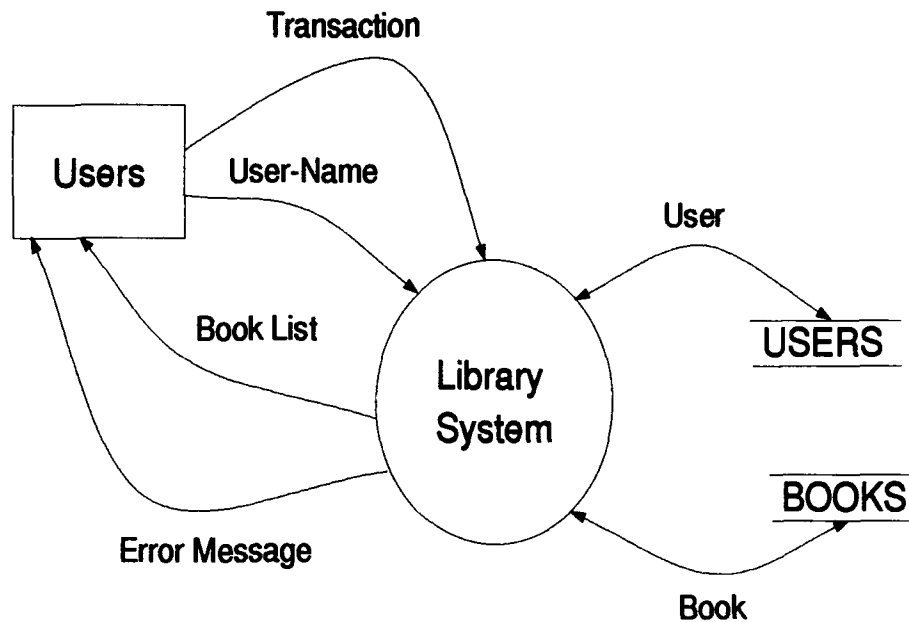


Figure 26. Library: Context Diagram (6:F-8)

The context diagram, Figure 26, shows the system's interaction between Users and the stores of books and authorized users. This interaction is expanded in Figure 27. All data flows that have a missing terminator are assumed to be flowing from or to an external entity. We have adopted a notion from Rumbaugh that allows a process to search an entire store. These are shown as unlabelled flows. (23:127) Addition or deletion of an item from a store is shown by a singled-headed arrow flowing into a store. Double-headed arrows between a process and a store denote the retrieval, modification, and replacement of an item in a store. We have also required that all flows have unique names. This facilitates an automated translation without having to generate parameterized behaviors. Transactions and User-Names allow Process 1 of Figure 27 to determine authorized users and distinguish staff users from ordinary borrowers. Process 2 determines the type of staff-authorized transaction requested, queries stores, and produces the appropriate lists as output. Process 3 performs ordinary borrower transactions.

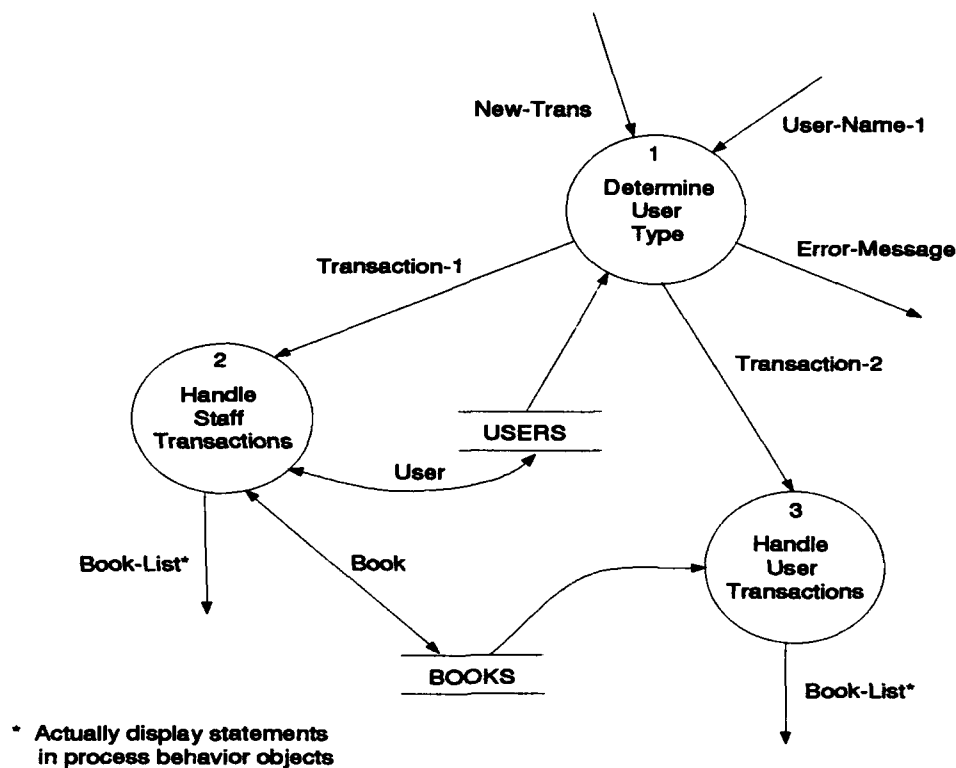


Figure 27. Library: Level 0

Figures 28, 29, and 30 expand Processes 1, 2, and 3 from the Level 0 DFD. Process 1 further decomposes into two lower-level processes: one to check the User data store and ensure that the user is authorized; the other to place controlling information in the transaction record. The transaction record is assumed to be input by the user and to contain all needed information to correctly complete the requested transaction. Process 1.1 produces an out-flow to notify a user who is not authorized access to the information in the system. Process 1.2 places the correct user level (staff or ordinary) in the transaction to restrict which transactions may be performed.

Process 2 decomposes into a transaction center and a set of processes representing each operation that is available to a staff user. Process 2.1 inspects TRANSACTION-1.trans-type and fills the appropriate out-flow, depending on its content. Processes 2.2 through 2.11 operate on information received in their Transaction flows. They also access and update stores as required by their function. For example, Process 2.4 (Check-Out Book) and Process 2.5 (Return Book) access

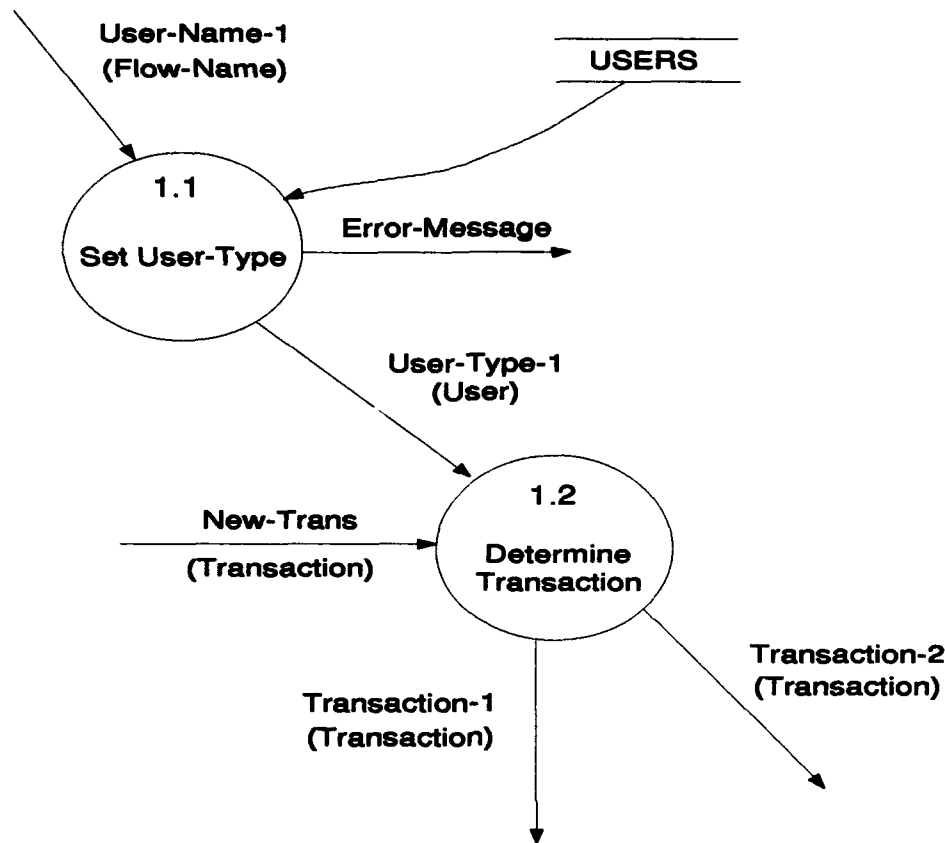


Figure 28. Library: Level 1

and update both Books and Users data stores. Each process must verify the status of the book being operated on, update the book's status to checked-out or available, and retrieve and update the number of books the user has checked out. Return Book must also update the last borrower of a returned book in the Book data store. Processes 2.6 through 2.9 produce out-flows of Book-List. Book-List is a set of books that would be displayed to the User. These are modeled in the processes' behaviors using OML's display function. No Flow objects named Book-List are required.

Process 3 also decomposes into transaction center and processes representing operations. Users have only one unique operation: listing books that they have borrowed. The other two operations shown are Level 2 processes, but have been included in Figure 30 for clarity.



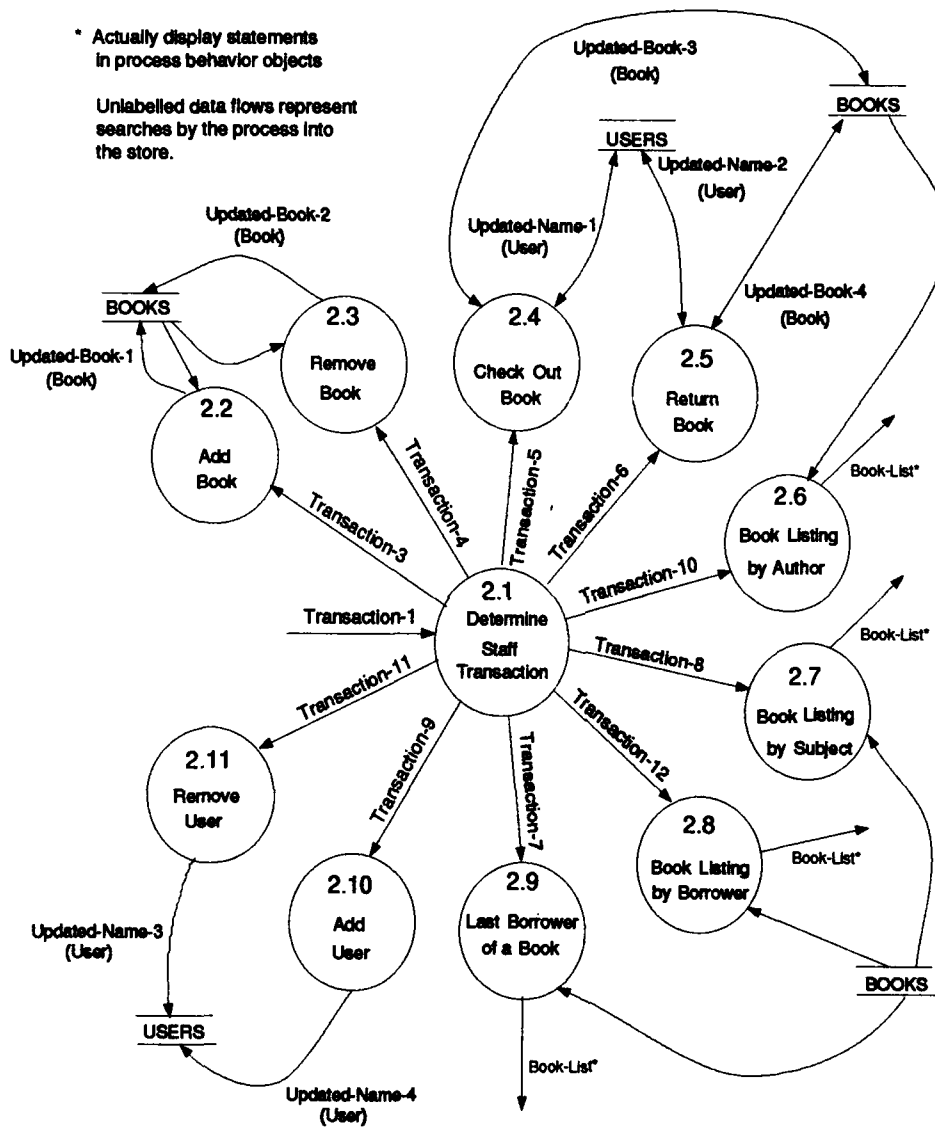


Figure 29. Library: Level 2 (6:F-10)

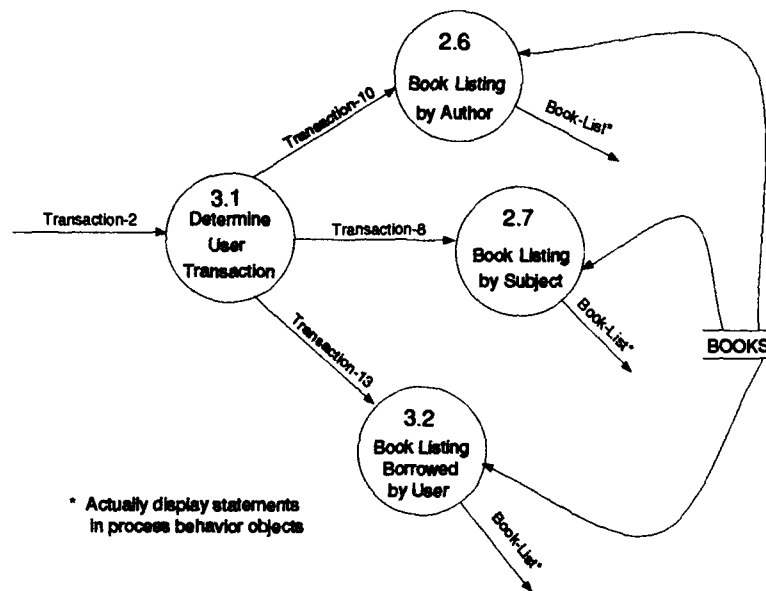


Figure 30. Library: Level 3 (6:F-11)

#### *E.4 Library Problem OML Specification*

This specification maps directly to OML from the models shown above with the following irregularities:

- In some cases, data flows to external entities have been modeled by a *display* rather than a flow when that flow was providing output information to the user. Book-List is an example of this type of data flow.
- The User entity class is defined in the ERM as having two subclasses. Because these subclasses had no unique attributes, they were differentiated using an attribute rather than creating subtypes and composing an ISA relation.
- Unlabeled arrows from stores to processes represent searches over the store to verify data. These flows are not modeled as OML flows because no data is ever removed from the store. They have been implemented by using OML's "exists" in the preconditions of the behavior's OML specification.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%                                                                 %%%
%%% File-Name : l.spec (Library Specification)                    %%%
%%%                                                                 %%%
%%% Authors : Capt Mary Boom, Capt Brad Mallare                  %%%
%%%                                                                 %%%
%%% Purpose : OML specification for the library problem.          %%%
%%%                                                                 %%%
%%% Unified Abstract Model Components :                            %%%
%%%   Entities, Relationships, Processes, Flows, Stores, Behaviors, and %%%
%%%   Relation-Tables                                             %%%
%%%                                                                 %%%
%%% Operation : After loading the translation code (trans-oml.fasl4) and %%%
%%% all the other code that it is dependent on, this OML specification %%%
%%% can be translated into an executable specification by typing the %%%
%%% following command at the Refine prompt:                        %%%
%%%                                                                 %%%
%%%           (convert "<your-OML-file-name>")                    %%%
%%%                                                                 %%%
%%% The name of the generated executable specification will be displayed %%%
%%% on the screen.  Additionally, the executable specification will be %%%
%%% automatically compiled and loaded.                             %%%
%%%                                                                 %%%
%%% After this file is translated, compiled, and loaded into Refine, %%%
%%% it can be executed by typing the following command at the Refine %%%
%%% prompt:                                                        %%%
%%%                                                                 %%%
%%%           (sim)                                                %%%
%%%                                                                 %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

specification library

%%%%%%%%%% ENTITIES %%%%%%%%%%

USER class-of entity

type : internal

parts

user-name : string;

kind : symbol range {staff, ordinary};

book-count : integer range {0 .. 10}

LIBRARIAN instance-of user

values

user-name : "librarian";

kind : staff;

book-count : 0

BOOM instance-of user

values

user-name : "boom";

kind : ordinary;

```

    book-count : 1

MALLARE instance-of user
values
    user-name : "mallare";
    kind : ordinary;
    book-count : 1

BOOK class-of entity
    type : internal
parts
    book-id : string;
    author : string;
    title : string;
    subject : string;
    last-user : string;
    current-borrower : string;
    status : symbol range {available, checked-out}

BOOK1 instance-of book
values
    book-id : "QA76.1";
    author : Ritchie;
    title : "The C Programming Language";
    subject : "programming";
    last-user : "none";
    current-borrower : "Boom";
    status : checked-out

BOOK2 instance-of book
values
    book-id : "QA76.2";
    author : "Silberschatz";
    title : "Operating System Concepts";
    subject : "operating systems";
    last-user : "none";
    current-borrower : "Mallare";
    status : checked-out

OUTSIDE instance-of entity
    type : external

FLOW-NAME class-of entity
    type : internal
parts
    N : string

TRANSACTION class-of entity
    type : internal
parts
    user-kind : symbol range {staff, ordinary};

```

```

    borrower-name : string;
    borrower-kind : symbol range {staff, ordinary};
% book-name : symbol;
    book-id : string;
    author : string;
    title : string;
    subject : string;
    trans-type : symbol range {add-book, remove-book, check-out-book,
                                return-book, list-books-by-author,
                                list-books-by-subject,
                                list-books-by-borrower, list-my-books,
                                list-last-borrower, add-user, remove-user}

```

```

MESSAGE class-of entity
    type : internal
parts
    text : string

```

```

ERROR-MESSAGE instance-of MESSAGE
values
    text : "Unauthorized User. See Librarian for assistance."

```

# %%%%%%%%%% RELATIONSHIPS %%%%%%%%%%%

```

Adds instance-of Relationship
    type : general
    cardinality : 1-1

```

```

Removes instance-of Relationship
    type : general
    cardinality : 1-1

```

```

Checks-Out instance-of Relationship
    type : general
    cardinality : 1-1

```

```

Returns instance-of Relationship
    type : general
    cardinality : 1-1

```

```

Gets-Subject-Listing instance-of Relationship
    type : general
    cardinality : 1-1

```

```

Gets-Author-Listing instance-of Relationship
    type : general
    cardinality : 1-1

```

```

Lists-Last-Borrower instance-of Relationship
    type : general
    cardinality : 1-1

```

Lists-All-Borrowed instance-of Relationship  
type : general  
cardinality : 1-m

List-Own-Borrowed instance-of Relationship  
type : general  
cardinality : 1-m

Add-Book-Behavior instance-of Relationship  
type : ICO  
cardinality : 1-1

%%%%%%%%%%%% PROCESSES %%%%%%%%%%%%%%

SET-USER-TYPE instance-of Process	%% 1.1
DETERMINE-TRANS-TYPE instance-of Process	%% 1.2
DETERMINE-STAFF-TRANS instance-of Process	%% 2.1
ADD-BOOK instance-of Process	%% 2.2
REMOVE-BOOK instance-of Process	%% 2.3
CHECK-OUT-BOOK instance-of Process	%% 2.4
RETURN-BOOK instance-of Process	%% 2.5
LIST-BOOKS-BY-AUTHOR instance-of Process	%% 2.6
LIST-BOOKS-BY-SUBJECT instance-of Process	%% 2.7
LIST-BOOKS-BY-BORROWER instance-of Process	%% 2.8
LIST-LAST-BORROWER instance-of Process	%% 2.9
ADD-USER instance-of Process	%% 2.10
REMOVE-USER instance-of Process	%% 2.11
DETERMINE-USER-TRANS instance-of Process	%% 3.1
LIST-MY-BOOKS instance-of Process	%% 3.2

%%%%%%%%%%%% BEHAVIORS %%%%%%%%%%%%%%

%% LEVEL 1 %%

SETTING-USER-TYPE instance-of Behavior

exists (User) (User in Users &

```

        User.UserName = User-Name-1.N &
        User.kind = staff)
-->
User-Type-1.kind := staff
event none;

exists (User) (User in Users &
        User.UserName = User-Name-1.N &
        User.kind = ordinary)
-->
User-Type-1.kind := ordinary &
User-Type-1.UserName := User-Name-1.N
event none;

not exists (User) (User in Users &
        User.UserName = User-Name-1.N)
-->
Display(Error-Message.Text)
event none

```

#### DETERMINE-TRANSACTION instance-of Behavior

```

User-Type-1.kind = staff
-->
Transaction-1 := New-Trans &
Transaction-1.user-kind := staff
event none;

User-Type-1.kind = ordinary
-->
Transaction-2 := New-Trans &
Transaction-2.user-kind := ordinary &
Transaction-2.Borrower-Name := User-Type-1.UserName
event none

```

#### %%% LEVEL 2 %%%

#### DETERMINING-STAFF instance-of Behavior

```

Transaction-1.trans-type = add-book
-->
Transaction-3 := Transaction-1
event none;

Transaction-1.trans-type = remove-book
-->
Transaction-4 := Transaction-1
event none;

Transaction-1.trans-type = check-out-book
-->

```



```

Transaction-5 := Transaction-1
event none;

Transaction-1.trans-type = return-book
-->
Transaction-6 := Transaction-1
event none;

Transaction-1.trans-type = list-books-by-author
-->
Transaction-10 := Transaction-1
event none;

Transaction-1.trans-type = list-books-by-subject
-->
Transaction-8 := Transaction-1
event none;

Transaction-1.trans-type = list-books-by-borrower
-->
Transaction-12 := Transaction-1
event none;

Transaction-1.trans-type = list-last-borrower
-->
Transaction-7 := Transaction-1
event none;

Transaction-1.trans-type = add-user
-->
Transaction-9 := Transaction-1
event none;

Transaction-1.trans-type = remove-user
-->
Transaction-11 := Transaction-1
event none

ADDING-BOOK instance-of Behavior
not exists (book) (book in Books &
                book.Book-Id = Transaction-3.Book-Id)
-->
% Updated-Book-1.Name := Transaction-3.book-name &
Updated-Book-1.book-id := Transaction-3.book-id &
Updated-Book-1.author := Transaction-3.author &
Updated-Book-1.title := Transaction-3.title &
Updated-Book-1.subject := Transaction-3.subject &
Updated-Book-1.last-user := "none" &
Updated-Book-1.current-borrower := "none" &
Updated-Book-1.status := available &
Books := Books union Updated-Book-1                %%set addition

```

```

event none

REMOVING-BOOK instance-of Behavior
exists (book) (book in Books &
               book.Book-Id = Transaction-4.Book-Id &
               Book.status = available)

-->
Books := Books set-diff {Book | book in Books &
                        book.book-id = Transaction-4.Book-Id} %%set removal

event none

CHECKING-BOOK-OUT instance-of Behavior %% Book-name are names of objects.
exists (book) (book in Books &
               book.Book-Id = Transaction-5.Book-Id &
               Book.status = available) &
exists (user) (user in Users &
               user.User-Name = Transaction-5.Borrower-Name &
               user.book-count < 10)

-->
Updated-Book-3 := getitem({Book | book in Books &
                          book.book-id = Transaction-5.Book-id}) &
Updated-Book-3.status := checked-out &
Updated-Book-3.current-borrower := Transaction-5.Borrower-Name &
% Books := Books union Updated-Book-3 &
Updated-Name-1 := getitem({User | user in Users &
                           user.User-Name = Transaction-5.Borrower-Name}) &
Updated-Name-1.book-count := Updated-Name-1.book-count + 1
% Users := Users union Updated-Name-1
event none

RETURNING-BOOK instance-of Behavior
exists (book) (book in Books &
               book.Book-Id = Transaction-6.Book-Id &
               Book.status = checked-out)

-->
Updated-Book-4 := getitem({Book | book in Books &
                          book.book-id = Transaction-6.Book-id}) &
Updated-Book-4.status := available &
Updated-Book-4.last-user := Transaction-6.Borrower-Name &
Updated-Book-4.current-borrower := "none" &
% Books := Books union Updated-Book-4 &
Updated-Name-2 := getitem({User | user in Users &
                           user.User-Name = Transaction-6.Borrower-Name}) &
Updated-Name-2.book-count := Updated-Name-2.book-count - 1
% Users := Users union Updated-Name-2
event none

LISTING-BY-AUTHOR instance-of Behavior
true
-->
Display({Book | Book in Books &

```

```

                                book.author = Transaction-10.author})
event none

LISTING-BY-SUBJECT instance-of Behavior
true
-->
Display({Book | book in books &
                                book.subject = Transaction-8.subject})
event none

LISTING-BY-BORROWER instance-of Behavior
exists (user) (user in Users &
                user.User-Name = Transaction-12.Borrower-Name)
-->
Display({Book | book in books &
                book.current-borrower = Transaction-12.Borrower-Name &
                book.status = checked-out})
event none

LISTING-LAST-BORROWER instance-of Behavior
exists (book) (book in Books &
                book.Book-Id = Transaction-7.Book-Id)
-->
Display({Book | book in books &
                Book.last-user = Transaction-7.Borrower-Name})
event none

ADDING-USER instance-of Behavior
not exists (user) (user in Users &
                user.User-Name = Transaction-9.Borrower-Name)
-->
Updated-Name-3.user-name := Transaction-9.borrower-name &
Updated-Name-3.kind := Transaction-9.borrower-kind &
Updated-Name-3.book-count := 0 &
Users := Users union Updated-Name-3      %% Set addition
event none

REMOVING-USER instance-of Behavior
exists (user) (user in Users &
                user.User-Name = Transaction-11.Borrower-Name &
                user.book-count = 0)
-->
Users := Users set-diff {user | user in Users &
                user.user-name = Transaction-11.Borrower-Name}  %% Set removal
event none

%%% LEVEL 3 %%%

DETERMINING-USER instance-of Behavior

Transaction-2.trans-type = list-books-by-author

```

```

-->
Transaction-10 := Transaction-2
event none;

Transaction-2.trans-type = list-books-by-subject
-->
Transaction-8 := Transaction-2
event none;

Transaction-2.trans-type = list-my-books
-->
Transaction-13 := Transaction-2
event none

LISTING-BORROWED-BY-USER instance-of Behavior
true
-->
Display({Book | book in books &
        book.current-borrower = Transaction-13.Borrower-Name})
event none

%%%%%%%%%%%% FLOWS %%%%%%%%%%

USER-NAME-1 instance-of Flow
flow-link : entity-proc
flow-data : Flow-Name

NEW-TRANS instance-of Flow
flow-link : entity-proc
flow-data : Transaction

USER-TYPE-1 instance-of Flow
flow-link : proc-proc
flow-data : User

TRANSACTION-1 instance-of Flow
flow-link : proc-proc
flow-data : Transaction

TRANSACTION-2 instance-of Flow
flow-link : proc-proc
flow-data : Transaction

UPDATED-BOOK-1 instance-of Flow
flow-link : proc-store
flow-data : Book

TRANSACTION-3 instance-of Flow
flow-link : proc-proc
flow-data : Transaction

```

UPDATED-BOOK-2 instance-of Flow  
flow-link : proc-store  
flow-data : Book

TRANSACTION-4 instance-of Flow  
flow-link : proc-proc  
flow-data : Transaction

UPDATED-NAME-1 instance-of Flow  
flow-link : proc-store  
flow-data : User

UPDATED-BOOK-3 instance-of Flow  
flow-link : proc-store  
flow-data : Book

TRANSACTION-5 instance-of Flow  
flow-link : proc-proc  
flow-data : Transaction

UPDATED-NAME-2 instance-of Flow  
flow-link : proc-store  
flow-data : User

UPDATED-BOOK-4 instance-of Flow  
flow-link : proc-store  
flow-data : Book

TRANSACTION-6 instance-of Flow  
flow-link : proc-proc  
flow-data : Transaction

TRANSACTION-7 instance-of Flow  
flow-link : proc-proc  
flow-data : Transaction

TRANSACTION-8 instance-of Flow  
flow-link : proc-proc  
flow-data : Transaction

TRANSACTION-9 instance-of Flow  
flow-link : proc-proc  
flow-data : Transaction

TRANSACTION-10 instance-of Flow  
flow-link : proc-proc  
flow-data : Transaction

TRANSACTION-11 instance-of Flow  
flow-link : proc-proc

```

flow-data : Transaction

TRANSACTION-12 instance-of Flow
  flow-link : proc-proc
  flow-data : Transaction

TRANSACTION-13 instance-of Flow
  flow-link : proc-proc
  flow-data : Transaction

UPDATED-NAME-3 instance-of Flow
  flow-link : proc-store
  flow-data : User

UPDATED-NAME-4 instance-of Flow
  flow-link : proc-store
  flow-data : User

ERROR-MESSAGE-1 instance-of Flow
  flow-link : proc-entity
  flow-data : message

```

# %%%%%%%%%% STORES %%%%%%%%%%%

```

USERS instance-of Store
  nature : set
  content: user

BOOKS instance-of Store
  nature : set
  content: book

```

# %%%%%%%%%% RELATION-TABLE %%%%%%%%%%%

```

Entity-Relation instance-of Relation-Table

```

%%NOTE: These are associations between entity classes! This is a result of  
 %% the hierarchy

%% Entity	Association	Entity %%
USER,	ADDS,	USER;
USER,	REMOVES,	USER;
USER,	ADDS,	BOOK;
USER,	REMOVES,	BOOK;
USER,	CHECKS-OUT,	BOOK;
USER,	RETURNS,	BOOK;
USER,	GET-SUBJECT-LISTING,	BOOK;
USER,	GET-AUTHOR-LISTING,	BOOK;
USER,	LISTS-LAST-BORROWED,	BOOK;
USER,	LISTS-ALL-BORROWED,	BOOK;

USER,               LISTS-OWN-BORROWED,       BOOK;

%Process-Behavior instance-of Relation-Table

%% Process	Assoc	Behavior	%%
DETERMINE-TRANS-TYPE,	ICO,	DETERMINE-TRANSACTION;	
SET-USER-TYPE,	ICO,	SETTING-USER-TYPE;	
DETERMINE-STAFF-TRANS,	ICO,	DETERMINING-STAFF;	
ADD-BOOK,	ICO,	ADDING-BOOK;	
REMOVE-BOOK,	ICO,	REMOVING-BOOK;	
CHECK-OUT-BOOK,	ICO,	CHECKING-BOOK-OUT;	
RETURN-BOOK,	ICO,	RETURNING-BOOK;	
LIST-BOOKS-BY-AUTHOR,	ICO,	LISTING-BY-AUTHOR;	
LIST-BOOKS-BY-SUBJECT,	ICO,	LISTING-BY-SUBJECT;	
LIST-BOOKS-BY-BORROWER,	ICO,	LISTING-BY-BORROWER;	
LIST-LAST-BORROWER,	ICO,	LISTING-LAST-BORROWER;	
LIST-MY-BOOKS,	ICO,	LISTING-BORROWED-BY-USER;	
ADD-USER,	ICO,	ADDING-USER;	
REMOVE-USER,	ICO,	REMOVING-USER;	
DETERMINE-USER-TRANS,	ICO,	DETERMINING-USER;	

%Flow-type-things instance-of Relation-Table

%% Object	Flow	Object
%LEVEL 1		
OUTSIDE,	USER-NAME-1,	SET-USER-TYPE;
OUTSIDE,	NEW-TRANS,	DETERMINE-TRANS-TYPE;
SET-USER-TYPE,	ERROR-MESSAGE-1,	OUTSIDE;
SET-USER-TYPE,	USER-TYPE-1,	DETERMINE-TRANS-TYPE;
DETERMINE-TRANS-TYPE,	TRANSACTION-1,	DETERMINE-STAFF-TRANS;
DETERMINE-TRANS-TYPE,	TRANSACTION-2,	DETERMINE-USER-TRANS;
%LEVEL 2		
ADD-BOOK,	UPDATED-BOOK-1,	BOOKS;
DETERMINE-STAFF-TRANS,	TRANSACTION-3,	ADD-BOOK;
REMOVE-BOOK,	UPDATED-BOOK-2,	BOOKS;
DETERMINE-STAFF-TRANS,	TRANSACTION-4,	REMOVE-BOOK;
CHECK-OUT-BOOK,	UPDATED-NAME-1,	USERS;
CHECK-OUT-BOOK,	UPDATED-BOOK-3,	BOOKS;
DETERMINE-STAFF-TRANS,	TRANSACTION-5,	CHECK-OUT-BOOK;
RETURN-BOOK,	UPDATED-NAME-2,	USERS;
RETURN-BOOK,	UPDATED-BOOK-4,	BOOKS;
DETERMINE-STAFF-TRANS,	TRANSACTION-6,	RETURN-BOOK;
DETERMINE-STAFF-TRANS,	TRANSACTION-7,	LIST-LAST-BORROWER;
DETERMINE-USER-TRANS,	TRANSACTION-8,	LIST-BOOKS-BY-SUBJECT;

DETERMINE-STAFF-TRANS, TRANSACTION-8,	LIST-BOOKS-BY-SUBJECT;
DETERMINE-USER-TRANS, TRANSACTION-10,	LIST-BOOKS-BY-AUTHOR;
DETERMINE-STAFF-TRANS, TRANSACTION-10,	LIST-BOOKS-BY-AUTHOR;
DETERMINE-STAFF-TRANS, TRANSACTION-12,	LIST-BOOKS-BY-BORROWER;
DETERMINE-USER-TRANS, TRANSACTION-13,	LIST-MY-BOOKS;
DETERMINE-STAFF-TRANS, TRANSACTION-9,	ADD-USER;
ADD-USER, UPDATED-NAME-3,	USERS;
DETERMINE-STAFF-TRANS, TRANSACTION-11,	REMOVE-USER;
REMOVE-USER, UPDATED-NAME-4,	USERS



### *E.5 Library Problem REFINE Executable Specification*

```
!! in-package ('RU)
!! in-grammar ('user)

var OML-Obj : object-class subtype-of user-object

var LIBRARY : object-class subtype-of OML-Obj

type return-values = tuple(validity: symbol,
                           next-procs : seq(symbol))

%%% Define object classes

var MESSAGE : object-class subtype-of LIBRARY
var MESSAGE-TEXT: map(MESSAGE, string) = {}

var TRANSACTION : object-class subtype-of LIBRARY
var TRANSACTION-USER-KIND: map(TRANSACTION, symbol) = {}
var TRANSACTION-BORROWER-NAME: map(TRANSACTION, string) = {}
var TRANSACTION-BORROWER-KIND: map(TRANSACTION, symbol) = {}
var TRANSACTION-BOOK-ID: map(TRANSACTION, string) = {}
var TRANSACTION-AUTHOR: map(TRANSACTION, string) = {}
var TRANSACTION-TITLE: map(TRANSACTION, string) = {}
var TRANSACTION-SUBJECT: map(TRANSACTION, string) = {}
var TRANSACTION-TRANS-TYPE: map(TRANSACTION, symbol) = {}

var FLOW-NAME : object-class subtype-of LIBRARY
var FLOW-NAME-N: map(FLOW-NAME, string) = {}

var BOOK : object-class subtype-of LIBRARY
var BOOK-BOOK-ID: map(BOOK, string) = {}
var BOOK-AUTHOR: map(BOOK, string) = {}
var BOOK-TITLE: map(BOOK, string) = {}
var BOOK-SUBJECT: map(BOOK, string) = {}
var BOOK-LAST-USER: map(BOOK, string) = {}
var BOOK-CURRENT-BORROWER: map(BOOK, string) = {}
var BOOK-STATUS: map(BOOK, symbol) = {}

var USER : object-class subtype-of LIBRARY
var USER-USER-NAME: map(USER, string) = {}
var USER-KIND: map(USER, symbol) = {}
var USER-BOOK-COUNT: map(USER, integer) = {}

%%% Define instances of object classes

var ERROR-MESSAGE : MESSAGE =
  set-attrs(make-object('MESSAGE),
            'name, '*ERROR-MESSAGE,
            'MESSAGE-TEXT, "Unauthorized User. See Librarian for assistance.")

var OUTSIDE-ENTITY : object-class subtype-of LIBRARY
var OUTSIDE : OUTSIDE-ENTITY =
  set-attrs(make-object('OUTSIDE-ENTITY),
            'name, '*OUTSIDE)
```

```

var BOOK2 : BOOK =
  set-attrs(make-object('BOOK),
    'name, '*BOOK2,
    'BOOK-BOOK-ID, "QA76.2",
    'BOOK-AUTHOR, "Silberschatz",
    'BOOK-TITLE, "Operating System Concepts",
    'BOOK-SUBJECT, "operating systems",
    'BOOK-LAST-USER, "none",
    'BOOK-CURRENT-BORROWER, "Mallare",
    'BOOK-STATUS, 'CHECKED-OUT)

var BOOK1 : BOOK =
  set-attrs(make-object('BOOK),
    'name, '*BOOK1,
    'BOOK-BOOK-ID, "QA76.1",
    'BOOK-AUTHOR, 'RITCHIE,
    'BOOK-TITLE, "The C Programming Language",
    'BOOK-SUBJECT, "programming",
    'BOOK-LAST-USER, "none",
    'BOOK-CURRENT-BORROWER, "Boom",
    'BOOK-STATUS, 'CHECKED-OUT)

var MALLARE : USER =
  set-attrs(make-object('USER),
    'name, '*MALLARE,
    'USER-USER-NAME, "mallare",
    'USER-KIND, 'ORDINARY,
    'USER-BOOK-COUNT, 1)

var BOOM : USER =
  set-attrs(make-object('USER),
    'name, '*BOOM,
    'USER-USER-NAME, "boom",
    'USER-KIND, 'ORDINARY,
    'USER-BOOK-COUNT, 1)

var LIBRARIAN : USER =
  set-attrs(make-object('USER),
    'name, '*LIBRARIAN,
    'USER-USER-NAME, "librarian",
    'USER-KIND, 'STAFF,
    'USER-BOOK-COUNT, 0)

%%% Define Store Objects

var BOOKS : set(BOOK) = {x | (x : BOOK) BOOK(x)}

var USERS : set(USER) = {x | (x : USER) USER(x)}

%%% Define objects for each flow object

var ERROR-MESSAGE-1 : MESSAGE =
  set-attrs(make-object('MESSAGE),
    'name, '*ERROR-MESSAGE-1)

var UPDATED-NAME-4 : USER =

```

```

    set-attrs(make-object('USER),
              'name, '*UPDATED-NAME-4)

var UPDATED-NAME-3 : USER =
    set-attrs(make-object('USER),
              'name, '*UPDATED-NAME-3)

var TRANSACTION-13 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-13)

var TRANSACTION-12 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-12)

var TRANSACTION-11 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-11)

var TRANSACTION-10 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-10)

var TRANSACTION-9 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-9)

var TRANSACTION-8 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-8)

var TRANSACTION-7 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-7)

var TRANSACTION-6 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-6)

var UPDATED-BOOK-4 : BOOK =
    set-attrs(make-object('BOOK),
              'name, '*UPDATED-BOOK-4)

var UPDATED-NAME-2 : USER =
    set-attrs(make-object('USER),
              'name, '*UPDATED-NAME-2)

var TRANSACTION-5 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-5)

var UPDATED-BOOK-3 : BOOK =
    set-attrs(make-object('BOOK),
              'name, '*UPDATED-BOOK-3)

var UPDATED-NAME-1 : USER =

```

```

    set-attrs(make-object('USER),
              'name, '*UPDATED-NAME-1)

var TRANSACTION-4 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-4)

var UPDATED-BOOK-2 : BOOK =
    set-attrs(make-object('BOOK),
              'name, '*UPDATED-BOOK-2)

var TRANSACTION-3 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-3)

var UPDATED-BOOK-1 : BOOK =
    set-attrs(make-object('BOOK),
              'name, '*UPDATED-BOOK-1)

var TRANSACTION-2 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-2)

var TRANSACTION-1 : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*TRANSACTION-1)

var USER-TYPE-1 : USER =
    set-attrs(make-object('USER),
              'name, '*USER-TYPE-1)

var NEW-TRANS : TRANSACTION =
    set-attrs(make-object('TRANSACTION),
              'name, '*NEW-TRANS)

var USER-NAME-1 : FLOW-NAME =
    set-attrs(make-object('FLOW-NAME),
              'name, '*USER-NAME-1)

%%% Define functions for behavior objects

function LISTING-BORROWED-BY-USER() =
    let(return-symbol : symbol = undefined)

    ( (if true
        then
        (enumerate element over
        {BOOK | (BOOK) (BOOK in BOOKS) &
        (BOOK-CURRENT-BORROWER(BOOK) = TRANSACTION-BORROWER-NAME(TRANSACTION-13))} do
        format(true, "~\\pp\\ ", element))
    ));
    return-symbol

function DETERMINING-USER() =
    let(return-symbol : symbol = undefined)

```

```

    ( (if (TRANSACTION-TRANS-TYPE(TRANSACTION-2) = 'LIST-BOOKS-BY-AUTHOR)
        then
      (assign-object('*TRANSACTION-2,'*TRANSACTION-10,'TRANSACTION)
      )
    );

    (if (TRANSACTION-TRANS-TYPE(TRANSACTION-2) = 'LIST-BOOKS-BY-SUBJECT)
        then
      (assign-object('*TRANSACTION-2,'*TRANSACTION-8,'TRANSACTION)
      )
    );

    (if (TRANSACTION-TRANS-TYPE(TRANSACTION-2) = 'LIST-MY-BOOKS)
        then
      (assign-object('*TRANSACTION-2,'*TRANSACTION-13,'TRANSACTION)
      )
    );
  ));
  return-symbol

function REMOVING-USER() =
  let(return-symbol : symbol = undefined)

  ( (if (ex (USER)((USER in USERS) &
    (USER-USER-NAME(USER) = TRANSACTION-BORROWER-NAME(TRANSACTION-11)) &
    (USER-BOOK-COUNT(USER) = 0)))
    then
    (USERS <- (setdiff(USERS, {USER | (USER) (USER in USERS) &
      (USER-USER-NAME(USER) = TRANSACTION-BORROWER-NAME(TRANSACTION-11))})))
  ));
  return-symbol

function ADDING-USER() =
  let(return-symbol : symbol = undefined)

  ( (if ~((ex (USER)((USER in USERS) &
    (USER-USER-NAME(USER) = TRANSACTION-BORROWER-NAME(TRANSACTION-9))))
    then
    (USER-USER-NAME(UPDATED-NAME-3) <- TRANSACTION-BORROWER-NAME(TRANSACTION-9));
    (USER-KIND(UPDATED-NAME-3) <- TRANSACTION-BORROWER-KIND(TRANSACTION-9));
    (USER-BOOK-COUNT(UPDATED-NAME-3) <- 0);
    (USERS <- (USERS with copy-object(UPDATED-NAME-3)))
  ));
  return-symbol

function LISTING-LAST-BORROWER() =
  let(return-symbol : symbol = undefined)

  ( (if (ex (BOOK)((BOOK in BOOKS) &
    (BOOK-BOOK-ID(BOOK) = TRANSACTION-BOOK-ID(TRANSACTION-7))))
    then
    (enumerate element over
    {BOOK | (BOOK) (BOOK in BOOKS) &
      (BOOK-LAST-USER(BOOK) = TRANSACTION-BORROWER-NAME(TRANSACTION-7))} do
      format(true, "\\pp\\ ",element))
  ));
  return-symbol

```

```

function LISTING-BY-BORROWER() =
  let(return-symbol : symbol = undefined)

  ( (if (ex (USER))((USER in USERS) &
    (USER-USER-NAME(USER) = TRANSACTION-BORROWER-NAME(TRANSACTION-12))))
    then
  (enumerate element over
{BOOK | (BOOK) (BOOK in BOOKS) &
  (BOOK-CURRENT-BORROWER(BOOK) = TRANSACTION-BORROWER-NAME(TRANSACTION-12)) &
  (BOOK-STATUS(BOOK) = 'CHECKED-OUT)} do
  format(true, "~\\pp\\ ",element))
));
  return-symbol

function LISTING-BY-SUBJECT() =
  let(return-symbol : symbol = undefined)

  ( (if true
    then
  (enumerate element over
{BOOK | (BOOK) (BOOK in BOOKS) &
  (BOOK-SUBJECT(BOOK) = TRANSACTION-SUBJECT(TRANSACTION-8))} do
  format(true, "~\\pp\\ ",element))
));
  return-symbol

function LISTING-BY-AUTHOR() =
  let(return-symbol : symbol = undefined)

  ( (if true
    then
  (enumerate element over
{BOOK | (BOOK) (BOOK in BOOKS) &
  (BOOK-AUTHOR(BOOK) = TRANSACTION-AUTHOR(TRANSACTION-10))} do
  format(true, "~\\pp\\ ",element))
));
  return-symbol

function RETURNING-BOOK() =
  let(return-symbol : symbol = undefined)

  ( (if (ex (BOOK))((BOOK in BOOKS) &
    (BOOK-BOOK-ID(BOOK) = TRANSACTION-BOOK-ID(TRANSACTION-6)) &
    (BOOK-STATUS(BOOK) = 'CHECKED-OUT)))
    then
  (UPDATED-BOOK-4 <- (arb({BOOK | (BOOK) (BOOK in BOOKS) &
    (BOOK-BOOK-ID(BOOK) = TRANSACTION-BOOK-ID(TRANSACTION-6))}));
  (BOOK-STATUS(UPDATED-BOOK-4) <- 'AVAILABLE);
  (BOOK-LAST-USER(UPDATED-BOOK-4) <- TRANSACTION-BORROWER-NAME(TRANSACTION-6));
  (BOOK-CURRENT-BORROWER(UPDATED-BOOK-4) <- "none");
  (UPDATED-NAME-2 <- (arb({USER | (USER) (USER in USERS) &
    (USER-USER-NAME(USER) = TRANSACTION-BORROWER-NAME(TRANSACTION-6))}));
  (USER-BOOK-COUNT(UPDATED-NAME-2) <- (USER-BOOK-COUNT(UPDATED-NAME-2) - 1))
  ));
  return-symbol

```

```

function CHECKING-BOOK-OUT() =
  let(return-symbol : symbol = undefined)

  ( (if (ex (BOOK)((BOOK in BOOKS) &
    (BOOK-BOOK-ID(BOOK) = TRANSACTION-BOOK-ID(TRANSACTION-5)) &
    (BOOK-STATUS(BOOK) = 'AVAILABLE)))
    and (ex (USER)((USER in USERS) &
    (USER-USER-NAME(USER) = TRANSACTION-BORROWER-NAME(TRANSACTION-5)) &
    (USER-BOOK-COUNT(USER) < 10)))
    then
      (UPDATED-BOOK-3 <- (arb({BOOK | (BOOK) (BOOK in BOOKS) &
        (BOOK-BOOK-ID(BOOK) = TRANSACTION-BOOK-ID(TRANSACTION-5))})))
      (BOOK-STATUS(UPDATED-BOOK-3) <- 'CHECKED-OUT);
      (BOOK-CURRENT-BORROWER(UPDATED-BOOK-3) <- TRANSACTION-BORROWER-NAME(TRANSACTION-5));
      (UPDATED-NAME-1 <- (arb({USER | (USER) (USER in USERS) &
        (USER-USER-NAME(USER) = TRANSACTION-BORROWER-NAME(TRANSACTION-5))})))
      (USER-BOOK-COUNT(UPDATED-NAME-1) <- (USER-BOOK-COUNT(UPDATED-NAME-1) + 1))
    ));
  return-symbol

function REMOVING-BOOK() =
  let(return-symbol : symbol = undefined)

  ( (if (ex (BOOK)((BOOK in BOOKS) &
    (BOOK-BOOK-ID(BOOK) = TRANSACTION-BOOK-ID(TRANSACTION-4)) &
    (BOOK-STATUS(BOOK) = 'AVAILABLE)))
    then
      (BOOKS <- (setdiff(BOOKS, {BOOK | (BOOK) (BOOK in BOOKS) &
        (BOOK-BOOK-ID(BOOK) = TRANSACTION-BOOK-ID(TRANSACTION-4))})))
    ));
  return-symbol

function ADDING-BOOK() =
  let(return-symbol : symbol = undefined)

  ( (if ~(ex (BOOK)((BOOK in BOOKS) &
    (BOOK-BOOK-ID(BOOK) = TRANSACTION-BOOK-ID(TRANSACTION-3))))
    then
      (BOOK-BOOK-ID(UPDATED-BOOK-1) <- TRANSACTION-BOOK-ID(TRANSACTION-3));
      (BOOK-AUTHOR(UPDATED-BOOK-1) <- TRANSACTION-AUTHOR(TRANSACTION-3));
      (BOOK-TITLE(UPDATED-BOOK-1) <- TRANSACTION-TITLE(TRANSACTION-3));
      (BOOK-SUBJECT(UPDATED-BOOK-1) <- TRANSACTION-SUBJECT(TRANSACTION-3));
      (BOOK-LAST-USER(UPDATED-BOOK-1) <- "none");
      (BOOK-CURRENT-BORROWER(UPDATED-BOOK-1) <- "none");
      (BOOK-STATUS(UPDATED-BOOK-1) <- 'AVAILABLE);
      (BOOKS <- (BOOKS with copy-object(UPDATED-BOOK-1)))
    ));
  return-symbol

function DETERMINING-STAFF() =
  let(return-symbol : symbol = undefined)

  ( (if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'ADD-BOOK)
    then
      (assign-object('*TRANSACTION-1,'*TRANSACTION-3,'TRANSACTION)

```

```

)
);

(if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'REMOVE-BOOK)
    then
    (assign-object('*TRANSACTION-1,'*TRANSACTION-4,'TRANSACTION)
    )
);

(if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'CHECK-OUT-BOOK)
    then
    (assign-object('*TRANSACTION-1,'*TRANSACTION-5,'TRANSACTION)
    )
);

(if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'RETURN-BOOK)
    then
    (assign-object('*TRANSACTION-1,'*TRANSACTION-6,'TRANSACTION)
    )
);

(if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'LIST-BOOKS-BY-AUTHOR)
    then
    (assign-object('*TRANSACTION-1,'*TRANSACTION-10,'TRANSACTION)
    )
);

(if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'LIST-BOOKS-BY-SUBJECT)
    then
    (assign-object('*TRANSACTION-1,'*TRANSACTION-8,'TRANSACTION)
    )
);

(if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'LIST-BOOKS-BY-BORROWER)
    then
    (assign-object('*TRANSACTION-1,'*TRANSACTION-12,'TRANSACTION)
    )
);

(if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'LIST-LAST-BORROWER)
    then
    (assign-object('*TRANSACTION-1,'*TRANSACTION-7,'TRANSACTION)
    )
);

(if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'ADD-USER)
    then
    (assign-object('*TRANSACTION-1,'*TRANSACTION-9,'TRANSACTION)
    )
);

(if (TRANSACTION-TRANS-TYPE(TRANSACTION-1) = 'REMOVE-USER)
    then
    (assign-object('*TRANSACTION-1,'*TRANSACTION-11,'TRANSACTION)
    )
));

```



```

return-symbol

function DETERMINE-TRANSACTION() =
  let(return-symbol : symbol = undefined)

  ( (if (USER-KIND(USER-TYPE-1) = 'STAFF)
        then
          (assign-object('*NEW-TRANS,*TRANSACTION-1,'TRANSACTION)
);
    (TRANSACTION-USER-KIND(TRANSACTION-1) <- 'STAFF)
  );

  (if (USER-KIND(USER-TYPE-1) = 'ORDINARY)
      then
        (assign-object('*NEW-TRANS,*TRANSACTION-2,'TRANSACTION)
  );
    (TRANSACTION-USER-KIND(TRANSACTION-2) <- 'ORDINARY);
    (TRANSACTION-BORROWER-NAME(TRANSACTION-2) <- USER-USER-NAME(USER-TYPE-1))
  ));
  return-symbol

function SETTING-USER-TYPE() =
  let(return-symbol : symbol = undefined)

  ( (if (ex (USER)((USER in USERS) &
              (USER-USER-NAME(USER) = FLOW-NAME-N(USER-NAME-1)) &
              (USER-KIND(USER) = 'STAFF)))
      then
        (USER-KIND(USER-TYPE-1) <- 'STAFF)
  );

  (if (ex (USER)((USER in USERS) &
              (USER-USER-NAME(USER) = FLOW-NAME-N(USER-NAME-1)) &
              (USER-KIND(USER) = 'ORDINARY)))
      then
        (USER-KIND(USER-TYPE-1) <- 'ORDINARY);
        (USER-USER-NAME(USER-TYPE-1) <- FLOW-NAME-N(USER-NAME-1))
  );

  (if ~(ex (USER)((USER in USERS) &
                  (USER-USER-NAME(USER) = FLOW-NAME-N(USER-NAME-1))))
      then
        (format(true, "\\pp\\ ",MESSAGE-TEXT(ERROR-MESSAGE)))
  ));
  return-symbol

%%% Define function for each state object

%%% Define function for each process object

function LIST-MY-BOOKS(dowhat : symbol) : return-values =
  let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
      [<'TRANSACTION,*TRANSACTION-13>],
      ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
      [],
      intflows-valid : boolean = false,

```

```

    check-flow : object = undefined,
    return-tuple : return-values = <'invalid,[]>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)))));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
          undefined?(retrieve-attribute(check-flow, x)))) then
          format(true, "Enter data for ~A~%", name(check-flow));
          check-flow <- modify-object(check-flow));
        LISTING-BORROWED-BY-USER();

      (enumerate flow over concat(int-flow-set, ext-flow-set) do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (enumerate flow-attr over return-attribute-list(check-flow) do
          store-attribute(check-flow, flow-attr, undefined)));
      return-tuple <- <'valid, []>
    else
      format(true, "Process cannot be executed.
        All in-flows are not defined.~%");
      return-tuple <- <'invalid, []>)
    else
      if intflows-valid then return-tuple <- <'valid, []>
      else return-tuple <- <'invalid, []>);
  return-tuple

  %%%
function DETERMINE-USER-TRANS(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [<'TRANSACTION, '*TRANSACTION-2>],
  ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [],
  intflows-valid : boolean = false,
  check-flow : object = undefined,
  return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)))));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do

```

```

        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
            undefined?(retrieve-attribute(check-flow, x)))) then
            format(true, "Enter data for ~A~%", name(check-flow));
            check-flow <- modify-object(check-flow));
        DETERMINING-USER();

    (enumerate flow over concat(int-flow-set, ext-flow-set) do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (enumerate flow-attr over return-attribute-list(check-flow) do
            store-attribute(check-flow, flow-attr, undefined));
        return-tuple <- <'valid, [ 'LIST-MY BOOKS, 'LIST-BOOKS-BY-AUTHOR,
            'LIST-BOOKS-BY-SUBJECT]>

    else
        format(true, "Process cannot be executed.
            All in-flows are not defined.~%",);
        return-tuple <- <'invalid, []>

    else
        if intflows-valid then return-tuple <- <'valid, []>
        else return-tuple <- <'invalid, []>;
    return-tuple

    %%%
function REMOVE-USER(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [<'TRANSACTION, '*TRANSACTION-11>],
    ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
        [],
    intflows-valid : boolean = false,
    check-flow : object = undefined,
    return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
    else
        (enumerate flow over int-flow-set do
            check-flow <- find-object(flow.flow-type, flow.flow-name);
            (enumerate flow-attr over return-attribute-list(check-flow) do
                if defined?(retrieve-attribute(check-flow, flow-attr)) then
                    intflows-valid <- true));

        (if dowhat = 'execute then
            (if intflows-valid then      %% if valid, check ext inflows
                (enumerate flow over ext-flow-set do
                    check-flow <- find-object(flow.flow-type, flow.flow-name);
                    (if (ex (x) (x in return-attribute-list(check-flow) &
                        undefined?(retrieve-attribute(check-flow, x)))) then
                        format(true, "Enter data for ~A~%", name(check-flow));
                        check-flow <- modify-object(check-flow));
                    REMOVING-USER();

                (enumerate flow over concat(int-flow-set, ext-flow-set) do
                    check-flow <- find-object(flow.flow-type, flow.flow-name);
                    (enumerate flow-attr over return-attribute-list(check-flow) do
                        store-attribute(check-flow, flow-attr, undefined));
                    return-tuple <- <'valid, []>

            else

```

```

format(true, "Process cannot be executed.
                All in-flows are not defined.~%");
return-tuple <- <'invalid, []>)
else
  if intflows-valid then return-tuple <- <'valid, []>
  else return-tuple <- <'invalid, []>);
return-tuple

####
function ADD-USER(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [<'TRANSACTION, '*TRANSACTION-9>],
    ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
        [],
    intflows-valid : boolean = false,
    check-flow : object = undefined,
    return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
          undefined?(retrieve-attribute(check-flow, x)))) then
          format(true, "Enter data for ~A~%", name(check-flow));
          check-flow <- modify-object(check-flow)));
      ADDING-USER();

    (enumerate flow over concat(int-flow-set, ext-flow-set) do
      check-flow <- find-object(flow.flow-type, flow.flow-name);
      (enumerate flow-attr over return-attribute-list(check-flow) do
        store-attribute(check-flow, flow-attr, undefined)));
    return-tuple <- <'valid, []>
  else
    format(true, "Process cannot be executed.
                All in-flows are not defined.~%");
    return-tuple <- <'invalid, []>)
  else
    if intflows-valid then return-tuple <- <'valid, []>
    else return-tuple <- <'invalid, []>);
return-tuple

####
function LIST-LAST-BORROWER(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [<'TRANSACTION, '*TRANSACTION-7>],
    ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
        [],

```

```

    intflows-valid : boolean = false,
    check-flow : object = undefined,
    return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
          undefined?(retrieve-attribute(check-flow, x)))) then
          format(true, "Enter data for ~A~%", name(check-flow));
          check-flow <- modify-object(check-flow));
        LISTING-LAST-BORROWER());

      (enumerate flow over concat(int-flow-set, ext-flow-set) do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (enumerate flow-attr over return-attribute-list(check-flow) do
          store-attribute(check-flow, flow-attr, undefined)));
      return-tuple <- <'valid, []>
    else
      format(true, "Process cannot be executed.
        All in-flows are not defined.~%");
      return-tuple <- <'invalid, []>)
    else
      if intflows-valid then return-tuple <- <'valid, []>
      else return-tuple <- <'invalid, []>);
  return-tuple

  %%%
function LIST-BOOKS-BY-BORROWER(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [<'TRANSACTION, '*TRANSACTION-12>],
  ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [],
  intflows-valid : boolean = false,
  check-flow : object = undefined,
  return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows

```

```

        (enumerate flow over ext-flow-set do
          check-flow <- find-object(flow.flow-type, flow.flow-name);
          (if (ex (x) (x in return-attribute-list(check-flow) &
            undefined?(retrieve-attribute(check-flow, x)))) then
            format(true, "Enter data for ~A~%", name(check-flow));
            check-flow <- modify-object(check-flow));
          LISTING-BY-BORROWER();

(enumerate flow over concat(int-flow-set, ext-flow-set) do
  check-flow <- find-object(flow.flow-type, flow.flow-name);
  (enumerate flow-attr over return-attribute-list(check-flow) do
    store-attribute(check-flow, flow-attr, undefined));
  return-tuple <- <'valid, []>
else
  format(true, "Process cannot be executed.
              All in-flows are not defined.~%");
  return-tuple <- <'invalid, []>
else
  if intflows-valid then return-tuple <- <'valid, []>
  else return-tuple <- <'invalid, []>;
return-tuple

#####
function LIST-BOOKS-BY-SUBJECT(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [<'TRANSACTION, '*TRANSACTION-8>],
    ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [],
    intflows-valid : boolean = false,
    check-flow : object = undefined,
    return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
          undefined?(retrieve-attribute(check-flow, x)))) then
          format(true, "Enter data for ~A~%", name(check-flow));
          check-flow <- modify-object(check-flow));
        LISTING-BY-SUBJECT();

(enumerate flow over concat(int-flow-set, ext-flow-set) do
  check-flow <- find-object(flow.flow-type, flow.flow-name);
  (enumerate flow-attr over return-attribute-list(check-flow) do
    store-attribute(check-flow, flow-attr, undefined));
  return-tuple <- <'valid, []>
else

```

```

format(true, "Process cannot be executed.
                                All in-flows are not defined.~%");
return-tuple <- <'invalid, []>)
else
  if intflows-valid then return-tuple <- <'valid, []>
  else return-tuple <- <'invalid, []>);
return-tuple

####
function LIST-BOOKS-BY-AUTHOR(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [<'TRANSACTION, '*TRANSACTION-10>],
    ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
        [],
    intflows-valid : boolean = false,
    check-flow : object = undefined,
    return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
          undefined?(retrieve-attribute(check-flow, x)))) then
          format(true, "Enter data for ~A~%", name(check-flow));
          check-flow <- modify-object(check-flow));
        LISTING-BY-AUTHOR());

      (enumerate flow over concat(int-flow-set, ext-flow-set) do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (enumerate flow-attr over return-attribute-list(check-flow) do
          store-attribute(check-flow, flow-attr, undefined)));
        return-tuple <- <'valid, []>
      else
        format(true, "Process cannot be executed.
                                All in-flows are not defined.~%");
        return-tuple <- <'invalid, []>)
    else
      if intflows-valid then return-tuple <- <'valid, []>
      else return-tuple <- <'invalid, []>);
return-tuple

####
function RETURN-BOOK(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [<'TRANSACTION, '*TRANSACTION-6>],
    ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
        [],

```

```

    intflows-valid : boolean = false,
    check-flow : object = undefined,
    return-tuple : return-values = <'invalid,[]>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
          undefined?(retrieve-attribute(check-flow, x)))) then
          format(true, "Enter data for ~A~%", name(check-flow));
          check-flow <- modify-object(check-flow));
        RETURNING-BOOK();

      (enumerate flow over concat(int-flow-set, ext-flow-set) do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (enumerate flow-attr over return-attribute-list(check-flow) do
          store-attribute(check-flow, flow-attr, undefined)));
      return-tuple <- <'valid, []>
    else
      format(true, "Process cannot be executed.
        All in-flows are not defined.~%");
      return-tuple <- <'invalid, []>
    else
      if intflows-valid then return-tuple <- <'valid, []>
      else return-tuple <- <'invalid, []>;
    return-tuple

  %%%
function CHECK-OUT-BOOK(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [<'TRANSACTION, '*TRANSACTION-5>],
  ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [],
  intflows-valid : boolean = false,
  check-flow : object = undefined,
  return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows

```



```

        (enumerate flow over ext-flow-set do
          check-flow <- find-object(flow.flow-type, flow.flow-name);
          (if (ex (x) (x in return-attribute-list(check-flow) &
            undefined?(retrieve-attribute(check-flow, x)))) then
            format(true, "Enter data for ~A~%", name(check-flow));
            check-flow <- modify-object(check-flow));
          CHECKING-BOOK-OUT();

(enumerate flow over concat(int-flow-set, ext-flow-set) do
  check-flow <- find-object(flow.flow-type, flow.flow-name);
  (enumerate flow-attr over return-attribute-list(check-flow) do
    store-attribute(check-flow, flow-attr, undefined));
  return-tuple <- <'valid, []>
else
  format(true, "Process cannot be executed.
              All in-flows are not defined.~%",);
  return-tuple <- <'invalid, []>)
else
  if intflows-valid then return-tuple <- <'valid, []>
  else return-tuple <- <'invalid, []>);
return-tuple

%%%
function REMOVE-BOOK(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [<'TRANSACTION, '*TRANSACTION-4],
    ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
        [],
    intflows-valid : boolean = false,
    check-flow : object = undefined,
    return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
          undefined?(retrieve-attribute(check-flow, x)))) then
          format(true, "Enter data for ~A~%", name(check-flow));
          check-flow <- modify-object(check-flow));
        REMOVING-BOOK();

(enumerate flow over concat(int-flow-set, ext-flow-set) do
  check-flow <- find-object(flow.flow-type, flow.flow-name);
  (enumerate flow-attr over return-attribute-list(check-flow) do
    store-attribute(check-flow, flow-attr, undefined));
  return-tuple <- <'valid, []>
else
  else

```

```

format(true, "Process cannot be executed.
                                All in-flows are not defined.~%");
return-tuple <- <'invalid, []>)
else
  if intflows-valid then return-tuple <- <'valid, []>
  else return-tuple <- <'invalid, []>;
return-tuple

%***
function ADD-BOOK(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [<'TRANSACTION, '*TRANSACTION-3>],
    ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [],
    intflows-valid : boolean = false,
    check-flow : object = undefined,
    return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
          undefined?(retrieve-attribute(check-flow, x)))) then
          format(true, "Enter data for ~A~%", name(check-flow));
          check-flow <- modify-object(check-flow));
        ADDING-BOOK();

      (enumerate flow over concat(int-flow-set, ext-flow-set) do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (enumerate flow-attr over return-attribute-list(check-flow) do
          store-attribute(check-flow, flow-attr, undefined)));
        return-tuple <- <'valid, []>
      else
        format(true, "Process cannot be executed.
                                All in-flows are not defined.~%");
        return-tuple <- <'invalid, []>)
    else
      if intflows-valid then return-tuple <- <'valid, []>
      else return-tuple <- <'invalid, []>;
    return-tuple

%***
function DETERMINE-STAFF-TRANS(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [<'TRANSACTION, '*TRANSACTION-1>],
    ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
    [],

```

```

    intflows-valid : boolean = false,
    check-flow : object = undefined,
    return-tuple : return-values = <'invalid,[]>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)))));

  (if dowhat = 'execute then
    (if intflows-valid then      %% if valid, check ext inflows
      (enumerate flow over ext-flow-set do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (if (ex (x) (x in return-attribute-list(check-flow) &
          undefined?(retrieve-attribute(check-flow, x)))) then
          format(true, "Enter data for "A~%", name(check-flow));
          check-flow <- modify-object(check-flow));
          DETERMINING-STAFF();

      (enumerate flow over concat(int-flow-set, ext-flow-set) do
        check-flow <- find-object(flow.flow-type, flow.flow-name);
        (enumerate flow-attr over return-attribute-list(check-flow) do
          store-attribute(check-flow, flow-attr, undefined)));
      return-tuple <- <'valid, [ 'REMOVE-USER, 'ADD-USER,
        'LIST-BOOKS-BY-BORROWER, 'LIST-BOOKS-BY-AUTHOR,
        'LIST-BOOKS-BY-SUBJECT, 'LIST-LAST-BORROWER,
        'RETURN-BOOK, 'CHECK-OUT-BOOK, 'REMOVE-BOOK, 'ADD-BOOK]>

    else
      format(true, "Process cannot be executed.
        All in-flows are not defined.~%");
      return-tuple <- <'invalid, []>)
    else
      if intflows-valid then return-tuple <- <'valid,[]>
      else return-tuple <- <'invalid,[]>);
  return-tuple

  %%%
function DETERMINE-TRANS-TYPE(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [<'USER, '*USER-TYPE-1>],
  ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [<'TRANSACTION, '*NEW-TRANS>],
  intflows-valid : boolean = false,
  check-flow : object = undefined,
  return-tuple : return-values = <'invalid,[]>)

(if size(int-flow-set) = 0 then intflows-valid <- true
else
  (enumerate flow over int-flow-set do
    check-flow <- find-object(flow.flow-type, flow.flow-name);
    (enumerate flow-attr over return-attribute-list(check-flow) do
      if defined?(retrieve-attribute(check-flow, flow-attr)) then
        intflows-valid <- true)))));

```

```

(if dowhat = 'execute then
  (if intflows-valid then      %% if valid, check ext inflows
    (enumerate flow over ext-flow-set do
      check-flow <- find-object(flow.flow-type, flow.flow-name);
      (if (ex (x) (x in return-attribute-list(check-flow) &
        undefined?(retrieve-attribute(check-flow, x)))) then
        format(true, "Enter data for ~A~%", name(check-flow));
        check-flow <- modify-object(check-flow));
      DETERMINE-TRANSACTION();

    (enumerate flow over concat(int-flow-set, ext-flow-set) do
      check-flow <- find-object(flow.flow-type, flow.flow-name);
      (enumerate flow-attr over return-attribute-list(check-flow) do
        store-attribute(check-flow, flow-attr, undefined));
      return-tuple <- <'valid, [ 'DETERMINE-USER-TRANS, 'DETERMINE-STAFF-TRANS]>
    else
      format(true, "Process cannot be executed.
                    All in-flows are not defined.~%");
      return-tuple <- <'invalid, []>)
    else
      if intflows-valid then return-tuple <- <'valid, []>
      else return-tuple <- <'invalid, []>);
  return-tuple

  %%%
function SET-USER-TYPE(dowhat : symbol) : return-values =
let (int-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [],
  ext-flow-set : seq(tuple(flow-type : symbol, flow-name : symbol)) =
  [<'FLOW-NAME, '*USER-NAME-1>],
  intflows-valid : boolean = false,
  check-flow : object = undefined,
  return-tuple : return-values = <'invalid, []>)

(if size(int-flow-set) = 0 then intflows-valid <- true
  else
    (enumerate flow over int-flow-set do
      check-flow <- find-object(flow.flow-type, flow.flow-name);
      (enumerate flow-attr over return-attribute-list(check-flow) do
        if defined?(retrieve-attribute(check-flow, flow-attr)) then
          intflows-valid <- true));

    (if dowhat = 'execute then
      (if intflows-valid then      %% if valid, check ext inflows
        (enumerate flow over ext-flow-set do
          check-flow <- find-object(flow.flow-type, flow.flow-name);
          (if (ex (x) (x in return-attribute-list(check-flow) &
            undefined?(retrieve-attribute(check-flow, x)))) then
            format(true, "Enter data for ~A~%", name(check-flow));
            check-flow <- modify-object(check-flow));
          SETTING-USER-TYPE();

        (enumerate flow over concat(int-flow-set, ext-flow-set) do
          check-flow <- find-object(flow.flow-type, flow.flow-name);
          (enumerate flow-attr over return-attribute-list(check-flow) do

```

```

        store-attribute(check-flow, flow-attr, undefined));
    return-tuple <- <'valid, [ 'DETERMINE-TRANS-TYPE]>
else
    format(true, "Process cannot be executed.
                                All in-flows are not defined.~%");
    return-tuple <- <'invalid, []>)
else
    if intflows-valid then return-tuple <- <'valid, []>
    else return-tuple <- <'invalid, []>);
return-tuple

%%%
function sim() =
let (pfunction : return-values = undefined,
    done : boolean = false,
    reply : integer = undefined,
    test : return-values = undefined,
    valid-procs : seq(symbol) = [],
    init-procs : seq(symbol) = ['SET-USER-TYPE])

reply <- Make-Menu(init-procs,
    "Choose one of these processes to initialize the simulation:");
(if Reply <= size(init-procs) then
    pfunction <- funcall(init-procs(reply), 'execute);
    while ~done do
        valid-procs <- [];
        (if pfunction.validity = 'valid then
            (if size(pfunction.next-procs) > 0 then
                (enumerate proc over pfunction.next-procs do
                    test <- funcall(proc, 'check);
                    (if test.validity = 'valid then
                        valid-procs <- append(valid-procs, proc)));
            reply <- Make-Menu(valid-procs,
                "Select a process that may potentially execute at this point:");
            (if Reply <= size(valid-procs) then
                pfunction <- funcall(valid-procs(reply), 'execute)
            elseif Reply = size(valid-procs)+2 then
                done <- true)                %% selects quit
            else pfunction.next-procs <- init-procs)
        else                                %% not valid process
            done <- true))

%%% Defines function for erasing all objects in Refine's database.
%%% Execute this function before you reload this file if you do not use
%%% the convert process.

function clear-objects() =
    (enumerate obj over [obj | (obj : LIBRARY) LIBRARY(obj)] do
        erase-object(obj))

```

## *Appendix F. OML User's Manual*

### *F.1 Synopsis*

This guide describes how to generate an executable specification from an OML specification. OML was specifically designed to directly and intuitively model all components of ERMs, DFM's, and STMs in a formal language syntax. OML's formal language representation enables OML specifications to be automatically translated into an executable form.

### *F.2 Required Software*

The following software is required to support the translation of an OML specification into an executable REFINE specification. The order in which the software is compiled and loaded into the REFINE environment is important and should be performed in the following order:

Software	Function
DIALECT	REFINE's language manipulation tool
oml-dm.re	OML's domain model
oml-gm.re	OML's grammar
lisp-utilities.lisp	Support functions written in Lisp
read-utilities.re	I/O support functions
obj-utilities.re	Object manipulation support functions
modify-obj.re	Object modification support functions
r-lib.re	Runtime library functions
trans-oml.re	Translates OML spec into an executable

Table 6. Software Required to Support OML's Translation and Execution

We have developed a lisp function to automatically compile and load these files into REFINE. If desired, this function can be used by loading the file "init.lisp" and typing "(init)" at the REFINE

command line. All of the above mentioned files are located on the *hawkeye* server at the Air Force Institute of Technology.

### *F.3 Assumptions*

The OML parsing and translation software was developed using the REFINE environment. The REFINE environment is designed to execute on a UNIX workstation and uses GNU Emacs for its user interface. Therefore, it is assumed that both REFINE and Emacs are available. Further it is assumed that the user has some familiarity with REFINE and Emacs.

The OML specification must be developed prior to using the translation tool. The description of OML provided in Chapter IV and Appendix A, as well as the two sample problems provided in Appendices D and E, provide sufficient guidance for developing an OML specification.

The user's problem must be informally modeled in terms of ERMs, STMs, and DFMs prior to developing the OML specification. Also, once the executable specification has been generated, these diagrams should be available for reference while testing the executable specification.

### *F.4 Generating an Executable Specification*

Once the OML specification has been developed, the steps listed in Table 7 should be taken to convert it into an executable specification.

### *F.5 Using the Executable OML Specification*

By running the executable specification, the user will be able to validate that his informal specification correctly specifies his requirements. Inconsistencies between requirements in the informal specification will result in an error, and a supporting error message will be displayed. Incorrect requirements will result in the executable not behaving in the manner intended by the user. Corrections to these errors should be made to the OML specification (either manually or through the

Next Step	Action	Result
Initialize REFINE (REFINE is initialized from the Emacs editor)	Type emacs & (in cmdtool window), Type Esc X run-refine	Emacs window will appear,  Emacs window will split and REFINE will initialize on the right
Load the translation software	Type (Load "init.lisp"), Type (init)	Loads Dialect, Loads translation software, Loads runtime library functions, Loads lisp functions
Convert the OML specification into an executable REFINE specification	Type (convert "<your OML file name>")	Parses the OML spec into an AST, Translates the info in the AST into a REFINE executable specification, Compiles and Loads the Executable specification, Tells the user to type (sim) to begin the simulation

Table 7. How to Generate an Executable Specification

front end tool) and a new executable specification should be produced by converting the modified OML specification. The REFINE executable specification is not intended to be modified.

If your executable specification terminates because of an error in the specification, an error message will be displayed on the screen. Take note of the state or process that was executing just prior to the error. Use the information provided in the error message, your knowledge of the last process or state to execute, and your informal specification to locate the error in your specification.

Also note, if the executable specification prompts the user to enter character data (letters and words) into the system, future reference to these characters will be case sensitive.

Once an executable specification has been created, it can be executed at any time. The user does not have to execute it immediately. If the user wants to log out of his account and then execute



it at a later time, the executable does not have to be recreated. Prior to executing a specification, all of the software mentioned in Section F.2 must be loaded first *except* oml-dm.re, oml-gm.re, and trans-oml.re. These files do not have to be loaded unless the user intends to convert an OML specification into an executable specification.

### *F.6 Diagnosing Errors*

This section is intended to help the OML user correct the types of errors he is likely to encounter. Errors in the OML specification can be revealed at three different stages: during parse, during compilation of the executable, or during execution. These error messages were obtained by converting the file "test.spec". "Test.spec" has the original errors commented out and the corrected statements right below the error.

*F.6.1 Errors detected while parsing.* Syntax errors in the OML specification will be detected during this stage of the translation. These errors will be caught immediately by the OML compiler and REFINe's interactive mode will direct you to the exact location where the syntax error occurred. The user should refer to Appendix A for a complete description of OML's syntax.

*F.6.2 Errors detected during compilation.* The errors detected during the compilation of the executable specification are the result of semantic errors in the OML specification. The current translation software does not perform semantic checking. It assumes the user has written a semantically correct OML specification. Thus, either semantic checking should be added to the translation software or else semantic checking should be performed by the elicitation tool (not yet built).

These errors are revealed after the OML specification is converted into an executable specification. However, when the executable specification is compiled, REFINe may detect an error causing an error message to be displayed to the screen.

The following are examples:

```

      :
Type checking...succeeded.....
REFINE compiling MOTOR-ON...
.
Type checking...
Warning: 1 local type conflict detected:

At program part:    IGNITION = 'OFF
Tried to match type: OBJECT
                  with type: symbol

MOTOR-ON did not compile correctly.
      :
```

The above error resulted from an improper reference to the status attribute of the IGNITION entity. The correct semantic should be IGNITION.STATUS = 'OFF. Note: REFINe notifies the user that the error occurred while compiling the MOTOR-ON state function.

```

      :
Warning: IDLE did not link correctly. The unlinked reference is:
      FIVE-MINUTE-TIMER
      :
REFINE compiling IDLE...
.
Type checking...
Warning: Unknown variable FIVE-MINUTE-TIMER in
      FIVE-MINUTE-TIMER.STATUS

IDLE did not compile correctly.scavenging...done
      :
```

The above error resulted from using an incorrect variable name. REFINe notified the user that the error occurred in the IDLE state. In this case, FIVE-MINUTE-TIMER is not the correct name of an entity object. The correct name should have been FIVE-MIN-TIMER.

These are two examples of errors found during compilation of the executable spec. The file "test.spec" shows the original error and the corrected form.

*F.6.3 Errors revealed during execution.* These errors are detected by executing the specification. These errors reveal inconsistencies, incorrectness, or incompleteness in the user's OML specification.

```

      :
(sim)

The current state of the system is OFF
  VALID STATE SPACE
"Events that can occur:"
1 ) SWITCH-TURNED-ON
2 ) Continue
3 ) Quit
1
Error: attempt to call 'RE:*UNDEFINED*' which is an undefined function.

Restart actions (select using :continue):
0: prompt for a new function, instead of 'RE:*UNDEFINED*'.
      :

```

Here, the execution terminated immediately. When this type of error message occurs (RE:\*UNDEFINED\*), it generally means that information is missing from the specification. In this case, the system crashed because the OML specification did not have entries in the Relation Table to associate an external event with its behavior. This type of error should eventually be caught by the semantic checking prior to the specification's execution.

```

:
(sim)
The current state of the system is OFF
  VALID STATE SPACE
  "Events that can occur:"
  1 ) SWITCH-TURNED-ON
  2 ) Continue
  3 ) Quit
  1
The current state of the system is IDLE
  INVALID STATE SPACE
The system's current state space conflicts with
  the state space required to be in the above mentioned state. Here are the
  current attribute values in the system. Compare them with the required values
  specified in your specification to find the inconsistencies.
  *WATER.THERMOSTAT-TEMP : 60
  *AIR.THERMOSTAT-TEMP : 60
  *FIVE-SEC-TIMER.TIMER-STATUS : OFF
  *FIVE-MIN-TIMER.TIMER-STATUS : OFF
  *OIL-VALVE.VALVE-STATUS : CLOSED
  *WATER-VALVE.VALVE-STATUS : CLOSED
  *COMBUSTION-SENSOR.SENSOR-STATUS : SAFE
  *FUEL-SENSOR.SENSOR-STATUS : SAFE
  *CONTROLLER.CONTROLLER-ENTITY-TR : 70
  *CONTROLLER.CONTROLLER-ENTITY-TW : 180
  *IGNITION.IGNITION-ENTITY-STATUS : OFF
  *MOTOR.MOTOR-ENTITY-STATUS : OFF
  *MOTOR.MOTOR-ENTITY-SPEED : INADEQUATE
  *MASTER-SWITCH.MASTER-SWITCH-ENTITY-STATUS : ON
  :

```

These kinds of errors are a result of inconsistent requirements in the specification. In this case, quite a bit of guidance is given to the user. We are told that the error occurred while in the IDLE state. The problem occurred because the system's current air temperature was 60 degrees and the controller.tr temperature was 70, but in order to enter the IDLE state, IDLE's state space required the air temperature to be greater than controller.tr-2. Thus, the system has a contradiction. As it turned out, this was an unnecessary requirement and it was removed from the specification.

## F.7 Test Specification

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%% File-Name : test.spec (Home-Heater Specification)          %%%
%%%
%%%
%%% Authors : Capt Mary Boom, Capt Brad Mallare                %%%
%%%
%%% Purpose : This home heater specification shows the stepwise %%%
%%% corrections made as the translated OML specification was verified %%%
%%% through execution of the specification.                     %%%
%%%
%%% Unified Abstract Model Components :                          %%%
%%%   Entities, Relationships, States, Events, Behaviors, and  %%%
%%%   Relation-Tables                                           %%%
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

specification test-heater

### %%%% ENTITIES

SENSOR class-of entity

type : external

parts

status : symbol range {safe, unsafe}

FUEL-SENSOR instance-of SENSOR

values

status : safe

COMBUSTION-SENSOR instance-of SENSOR

values

status : safe

VALVE class-of entity

type : external

parts

status : symbol range {open, closed}

WATER-VALVE instance-of VALVE

values

status : closed

OIL-VALVE instance-of VALVE

values

status : closed

TIMER class-of entity

type : external

parts

```

    status : symbol range {off, on}

FIVE-MIN-TIMER instance-of TIMER
    values
    status : off

FIVE-SEC-TIMER instance-of TIMER
    values
    status : off

THERMOSTAT class-of entity
    type : external
    parts
        temp : integer range {0 .. 280}

AIR instance-of THERMOSTAT
    values
    temp : 60

WATER instance-of THERMOSTAT
    values
    temp : 60

MASTER-SWITCH instance-of entity
    type : external
    parts
        status : symbol range {on, off} init-val off

MOTOR instance-of entity
    type : external
    parts
        status : symbol range {on, off} init-val off;
        speed : symbol range {adequate, inadequate} init-val inadequate

IGNITION instance-of entity
    type : external
    parts
        status : symbol range {on, off} init-val off

CONTROLLER instance-of entity
    type : external
    parts
        tr : integer range {32 .. 130} init-val 70; %% preset air temp
        tw : integer range {32 .. 280} init-val 180 %% preset water temp

%%%% RELATIONSHIPS %%%%

ACTIVATES instance-of relationship
    type : general
    cardinality : 1-1

```

MONITORS instance-of relationship  
type : general  
cardinality : 1-1

CONTROLS-FLOW instance-of relationship  
type : general  
cardinality : 1-1

SWITCHES instance-of relationship  
type : general  
cardinality : 1-1

SIGNALS instance-of relationship  
type : general  
cardinality : 1-1

ICD instance-of relationship  
type : ICD  
cardinality : 1-1

%%%% STATES %%%%%

OFF instance-of state  
state-space : master-switch.status = off

IDLE instance-of state  
state-space : master-switch.status = on;  
% air.temp > controller.tr - 2; % unnecessary requirement  
% five-minute-timer.status = off; % incorrect name  
five-min-timer.status = off;  
five-sec-timer.status = off

MOTOR-ON instance-of state  
state-space : master-switch.status = on;  
motor.status = on;  
motor.speed = inadequate;  
air.temp < controller.tr - 2;  
% ignition = off; % incorrect attribute reference  
ignition.status = off;  
oil-valve.status = closed

WATER-HEATING instance-of state  
state-space : master-switch.status = on;  
air.temp < controller.tr + 2;  
motor.status = on;  
motor.speed = adequate;  
water.temp < controller.tw;  
fuel-sensor.status = safe;  
combustion-sensor.status = safe;  
water-valve.status = closed;

oil-valve.status = open

**RUNNING instance-of state**

state-space : master-switch.status = on;  
air.temp < controller.tr + 2;  
motor.status = on;  
motor.speed = adequate;  
water.temp > controller.tw;  
fuel-sensor.status = safe;  
combustion-sensor.status = safe;  
water-valve.status = open;  
oil-valve.status = open

**SHUTDOWN instance-of state**

state-space : master-switch.status = on;  
air.temp >= controller.tr + 2;  
motor.status = on;  
fuel-sensor.status = safe;  
combustion-sensor.status = safe;  
water-valve.status = open;  
oil-valve.status = closed;  
five-sec-timer.status = on

**WAIT5MINUTES instance-of state**

state-space : master-switch.status = on;  
motor.status = off;  
fuel-sensor.status = safe;  
combustion-sensor.status = safe;  
water-valve.status = closed;  
oil-valve.status = closed;  
ignition.status = off;  
five-sec-timer.status = off;  
five-min-timer.status = on

**HOLD instance-of state**

state-space : ignition.status = off;  
five-sec-timer.status = off;  
five-min-timer.status = off;  
motor.status = off;  
water-valve.status = closed;  
oil-valve.status = closed;

% combustion-sensor.status = unsafe; % master switch off & safe  
% fuel-sensor.status = unsafe % sensors also gets to here

**%%%% EVENTS %%%%**

**%% Internal Events**

**MASTER-SWITCH-ON instance-of event**

type: internal



MOTOR-TURNED-ON instance-of event  
type: internal

OIL-IGNITED instance-of event  
type: internal

WATER-VALVE-OPENED instance-of event  
type: internal

DONE-HEATING-WATER instance-of event  
type: internal

MOTOR-TURNED-OFF instance-of event  
type: internal

DONE-WAITING instance-of event  
type: internal

MASTER-SWITCH-OFF instance-of event  
type: internal

ABNORMAL-FUEL instance-of event  
type: internal

ABNORMAL-COMBUSTION instance-of event  
type: internal

SYSTEM-IS-RESET instance-of event  
type: internal

SYSTEM-IS-OFF instance-of event  
type: internal

#### %% External Events

SWITCH-TURNED-ON instance-of event  
type: external

AIR-TEMP-BELOW-PRESET instance-of event  
type: external

ADEQUATE-MOTOR-SPEED instance-of event  
type: external

WATER-TEMP-ABOVE-PRESET instance-of event  
type: external

AIR-TEMP-ABOVE-PRESET instance-of event  
type: external

FIVE-SEC-TIMER-EXPIRES instance-of event  
type: external

FIVE-MIN-TIMER-EXPIRES instance-of event  
type: external

SWITCH-TURNED-OFF instance-of event  
type: external

UNSAFE-COMBUSTION-SENSOR instance-of event  
type: external

UNSAFE-FUEL-SENSOR instance-of event  
type: external

RESET-SYSTEM instance-of event  
type: external

SYSTEM-TURNED-OFF instance-of event  
type: external

#### %%%%% BEHAVIORS %%%%%

FURNACE-OFF instance-of behavior  
master-switch.status, = on  
-->  
event, MASTER-SWITCH-ON

#### %%%%%

FURNACE-IDLE instance-of behavior

air.temp, < controller.tr - 2, dont-care;  
master-switch.status, = on, = off  
-->  
motor.status, on, off  
event, MOTOR-TURNED-ON, MASTER-SWITCH-OFF

#### %%%%%

FURNACE-MOTOR-ON instance-of behavior

motor.speed, dont-care, = adequate;  
master-switch.status, = off, = on  
-->  
ignition.status, off, on;  
oil-valve.status, closed, open;  
motor.status, off, on  
event, MASTER-SWITCH-OFF, OIL-IGNITED

%%%%

FURNACE-WATER-HEATING instance-of behavior

```
Water.temp,      > controller.tw,  dont-care,  dont-care,  dont-care;
master-switch.status,  = on,  = off,  dont-care,  dont-care;
fuel-sensor.status,    = safe, dont-care,  = unsafe,  dont-care;
combustion-sensor.status, = safe, dont-care,  dont-care,  = unsafe
-->
water-valve.status,    open,  closed,  closed,  closed;
oil-valve.status,      open,  closed,  closed,  closed;
five-sec-timer.status, off,  on,  on,  on
event, WATER-VALVE-OPENED, MASTER-SWITCH-OFF, ABNORMAL-FUEL,
                                           ABNORMAL-COMBUSTION
```

%%%%

FURNACE-RUNNING instance-of behavior

```
air.temp,      >= controller.tr + 2,  dont-care,  dont-care,  dont-care;
fuel-sensor.status,    = safe, = unsafe,  dont-care,  dont-care;
combustion-sensor.status, = safe, dont-care,  = unsafe,  dont-care;
master-switch.status,  = on,  = on,  = on,  = on
-->
oil-valve.status,      closed, closed,  closed,  closed;
five-sec-timer.status, on,  on,  on,  on
event,  DONE-HEATING-WATER, ABNORMAL-FUEL,  ABNORMAL-COMBUSTION,
                                           MASTER-SWITCH-OFF
```

%%%%

FURNACE-SHUTTING-DOWN instance-of behavior

```
five-sec-timer.status,  = off,  = off,  = off;
fuel-sensor.status,     = safe, = unsafe,  dont-care;
combustion-sensor.status, = safe, dont-care,  = unsafe
-->
motor.status,           off,  off,  off;
water-valve.status,     closed, closed,  closed;
five-min-timer.status,  on,  off,  off;
ignition.status,        off,  off,  off
event,  FIVE-SEC-TIMER-EXPIRES, ABNORMAL-SHUTDOWN, ABNORMAL-SHUTDOWN
```

%%%%

FURNACE-WAITING instance-of behavior

```
five-min-timer.status,  = off
-->
event,  FIVE-MIN-TIMER-EXPIRES
```

%%%%

FURNACE-ABNORMAL instance-of behavior

```
fuel-sensor.status,      = safe,      = safe;
combustion-sensor.status, = safe,      = safe;
master-switch.status,    = off,      = on
-->
event,                    SYSTEM-IS-OFF, SYSTEM-IS-RESET
```

%%%%%%%% BEHAVIORS - EVENT ACTIONS %%%%%%

SWITCH-TURNED-ON-BEH instance-of behavior

```
true
-->
master-switch.status := on
event none
```

AIR-TEMP-BELOW-PRESET-BEH instance-of behavior

```
true
-->
air.temp := controller.tr - 3
event none
```

ADEQUATE-MOTOR-SPEED-BEH instance-of behavior

```
true
-->
motor.speed := adequate
event none
```

WATER-TEMP-ABOVE-PRESET-BEH instance-of behavior

```
true
-->
water.temp := controller.tw + 1
event none
```

AIR-TEMP-ABOVE-PRESET-BEH instance-of behavior

```
true
-->
air.temp := controller.tr + 3
event none
```

FIVE-SEC-TIMER-EXPIRES-BEH instance-of behavior

```
true
-->
five-sec-timer.status := off
event none
```

FIVE-MIN-TIMER-EXPIRES-BEH instance-of behavior

```
true
-->
```

```

five-min-timer.status := off
event none

```

```

SWITCH-TURNED-OFF-BEH instance-of behavior
true
-->
master-switch.status := off
event none

```

```

UNSAFE-COMBUSTION-SENSOR-BEH instance-of behavior
true
-->
combustion-sensor.status := unsafe
event none

```

```

UNSAFE-FUEL-SENSOR-BEH instance-of behavior
true
-->
fuel-sensor.status := unsafe
event none

```

```

RESET-SYSTEM-BEH instance-of behavior
true
-->
fuel-sensor.status := safe &
combustion-sensor.status := safe &
master-switch.status := on
event none

```

```

SYSTEM-TURNED-OFF-BEH instance-of behavior
true
-->
fuel-sensor.status := safe &
combustion-sensor.status := safe &
master-switch.status := off
event none

```

TABLE1 instance-of relation-table

%%FROM-OBJECT	ASSOCIATION	TO-OBJECT
%% STATE-EVENT RELATIONSHIPS		
OFF,	MASTER-SWITCH-ON,	IDLE;
IDLE,	MOTOR-TURNED-ON,	MOTOR-ON;
MOTOR-ON,	OIL-IGNITED,	WATER-HEATING;
WATER-HEATING,	WATER-VALVE-OPENED,	RUNNING;
RUNNING,	DONE-HEATING-WATER,	SHUTDOWN;
SHUTDOWN,	MOTOR-TURNED-OFF,	WAIT5MINUTES;
WAIT5MINUTES,	DONE-WAITING,	IDLE;

IDLE,	MASTER-SWITCH-OFF,	OFF;
MOTOR-ON,	MASTER-SWITCH-OFF,	OFF;
WATER-HEATING,	MASTER-SWITCH-OFF,	SHUTDOWN;
WATER-HEATING,	ABNORMAL-FUEL,	SHUTDOWN;
WATER-HEATING,	ABNORMAL-COMBUSTION,	SHUTDOWN;
RUNNING,	MASTER-SWITCH-OFF,	SHUTDOWN;
RUNNING,	ABNORMAL-FUEL,	SHUTDOWN;
RUNNING,	ABNORMAL-COMBUSTION,	SHUTDOWN;
SHUTDOWN,	ABNORMAL-SHUTDOWN,	HOLD;
HOLD,	SYSTEM-IS-RESET,	IDLE;
HOLD,	SYSTEM-IS-OFF,	OFF;

#### %% STATE- EXTERNAL-EVENT RELATIONSHIPS

OUTSIDE,	SWITCH-TURNED-ON,	OFF;
OUTSIDE,	AIR-TEMP-BELOW-PRESET,	IDLE;
OUTSIDE,	ADEQUATE-MOTOR-SPEED,	MOTOR-ON;
OUTSIDE,	WATER-TEMP-ABOVE-PRESET,	WATER-HEATING;
OUTSIDE,	AIR-TEMP-ABOVE-PRESET,	RUNNING;
OUTSIDE,	FIVE-SEC-TIMER-EXPIRES,	SHUTDOWN;
OUTSIDE,	FIVE-MIN-TIMER-EXPIRES,	WAIT5MINUTES;
OUTSIDE,	SWITCH-TURNED-OFF,	IDLE;
OUTSIDE,	SWITCH-TURNED-OFF,	MOTOR-ON;
OUTSIDE,	SWITCH-TURNED-OFF,	WATER-HEATING;
OUTSIDE,	SWITCH-TURNED-OFF,	RUNNING;
OUTSIDE,	UNSAFE-COMBUSTION-SENSOR,	WATER-HEATING;
OUTSIDE,	UNSAFE-COMBUSTION-SENSOR,	RUNNING;
OUTSIDE,	UNSAFE-FUEL-SENSOR,	WATER-HEATING;
OUTSIDE,	UNSAFE-FUEL-SENSOR,	RUNNING;
OUTSIDE,	RESET-SYSTEM,	HOLD;
OUTSIDE,	SYSTEM-TURNED-OFF,	HOLD;

#### %% ENTITY-RELATIONSHIPS

CONTROLLER,	ACTIVATES,	MOTOR;
CONTROLLER,	MONITORS,	MOTOR;
CONTROLLER,	MONITORS,	THERMOSTAT;
CONTROLLER,	CONTROLS-FLOW,	VALVE;
MASTER-SWITCH,	SWITCHES,	CONTROLLER;
FUEL-SENSOR,	SIGNALS,	CONTROLLER;
COMBUSTION-SENSOR,	SIGNALS,	CONTROLLER;

#### %% EVENT-BEHAVIOR-RELATIONSHIPS

%% these realtion-table entries were initially missing, causing  
 %% the specification to terminate because external-events could not  
 %% be associated with their behaviors.

SWITCH-TURNED-ON,	ICO,	SWITCH-TURNED-ON-BEH;
AIR-TEMP-BELOW-PRESET,	ICO,	AIR-TEMP-BELOW-PRESET-BEH;
ADEQUATE-MOTOR-SPEED,	ICO,	ADEQUATE-MOTOR-SPEED-BEH;

WATER-TEMP-ABOVE-PRESET,	ICO,	WATER-TEMP-ABOVE-PRESET-BEH;
AIR-TEMP-ABOVE-PRESET,	ICO,	AIR-TEMP-ABOVE-PRESET-BEH;
FIVE-SEC-TIMER-EXPIRES,	ICO,	FIVE-SEC-TIMER-EXPIRES-BEH;
FIVE-MIN-TIMER-EXPIRES,	ICO,	FIVE-MIN-TIMER-EXPIRES-BEH;
SWITCH-TURNED-OFF,	ICO,	SWITCH-TURNED-OFF-BEH;
UNSAFE-COMBUSTION-SENSOR,	ICO,	UNSAFE-COMBUSTION-SENSOR-BEH;
UNSAFE-FUEL-SENSOR,	ICO,	UNSAFE-FUEL-SENSOR-BEH;
RESET-SYSTEM,	ICO,	RESET-SYSTEM-BEH;
SYSTEM-TURNED-OFF,	ICO,	SYSTEM-TURNED-OFF-BEH;

# %% STATE-BEHAVIOR RELATIONSHIPS

OFF,	ICO,	FURNACE-OFF ;
IDLE,	ICO,	FURNACE-IDLE;
MOTOR-ON,	ICO,	FURNACE-MOTOR-ON;
WATER-HEATING,	ICO,	FURNACE-WATER-HEATING;
RUNNING,	ICO,	FURNACE-RUNNING;
SHUTDOWN,	ICO,	FURNACE-SHUTTING-DOWN;
WAIT5MINUTES,	ICO,	FURNACE-WAITING;
HOLD,	ICO,	FURNACE-ABNORMAL

## Bibliography

1. Albrecht, Paul F. and others. "Source-to-Source Translation: Ada to Pascal and Pascal to Ada," *ACM SIGPLAN Notices*, 15(11):183-193 (November 1980).
2. Balzer, Robert. "Software Technology in the 1990's: Using a New Paradigm," *IEEE Computer*, 16(11):39-45 (November 1983).
3. Balzer, Robert and Neil Goldman. "Principles of Good Software Specification and their Implications for Specification Language," *IEEE Conference on Specifications of Reliable Software*, 7(2):58-67 (March 1979).
4. Berzins, Valdis and Luqi. "An Introduction to the Specification Language Spec," *IEEE Software*, 7(2):74-84 (March 1990).
5. Bjørner, Dines. "Programming in the Meta-Language: A Tutorial." *The Vienna Development Method: The Meta-Language* edited by Dines Bjørner and Cliff B. Jones, Springer-Verlag, 1978.
6. Blankenship, Captain Donald D. *Generalized Method for Transforming Informal Analysis Methods to Refine Formal Specifications*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
7. Boehm, Barry W. "Software Engineering," *IEEE Transactions on Computers*, C-25(12):1226-1240 (December 1976).
8. Buska, Douglas E. and Mark L. Wilkins. "ADL: A Dynamic Object-Oriented Modeling Language," *OOPS Messenger*, 2(1):8-27 (January 1991).
9. Davis, Alan A. *Software Requirements Analysis and Specification*. Englewood Cliffs, NJ: Prentice Hall, 1990.
10. Douglass, Captain Randal L. *Formalization and Executable Simulation of SADT Requirements Analysis Using the Refine Specification Language*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991. DTIC Number.
11. Feather, Martin S. "The Evolution of Composite System Specifications." *Fourth International Workshop on Software Specification and Design*. 52-57. Washington, D.C. 20036-1903: Computer Society Press of the IEEE, April 1987.
12. Fetzer, James H. "Program Verification: The Very Idea," *Communications of the ACM*, 31(9):1048-1063 (September 1988).
13. Fraser, Martin D., et al. "Informal and Formal Requirements Specification Languages: Bridging the Gap," *IEEE Transactions on Software Engineering*, 17(5):454-465 (May 1991).
14. Goldsack, S. J., editor. *Ada for Specification: Possibilities and Limitations*. The Ada Companion Series, New York, New York: Cambridge University Press, 1985.
15. Greenspan, Sol J. *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition*. PhD dissertation, The University of Toronto, Toronto, Canada, 1984.
16. Guttag, John V., et al. "The Larch Family of Specification Languages," *IEEE Software*, 2(5):24-36 (September 1985).
17. Hurley, Richard B. *Decision Tables in Software Engineering*. The Van Nostrand Reinhold Data Processing Series, New York, New York: Van Nostrand Reinhold Co., 1983.
18. IEEE Computer Society (LCRST, Japan) and Alvey Directorate (UK) ACM SIGSOFT and Agence de l'Informatique & AFCET (France). *Fourth International Workshop on Software Specification and Design*, Washington, D.C. 20036-1903: Computer Society Press of the IEEE, April 1987.



19. Jones, Cliff B. "The Meta-Language: A Reference Manual." *The Vienna Development Method: The Meta-Language* edited by Dines Bjørner and Cliff B. Jones, Springer-Verlag, 1978.
20. Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science, Englewood Cliffs, NJ: Prentice Hall, 1988.
21. Reasoning Systems Inc., Palo Alto, CA. *DIALECT<sup>TM</sup> User's Guide*. For DIALECT<sup>TM</sup> Version 1.0 on Sun Computers.
22. Reasoning Systems Inc., Palo Alto, CA. *REFINE<sup>TM</sup> User's Guide*. For REFINETM Version 3.0.
23. Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
24. Shlaer, Sally and Stephen J. Mellor. *Object-Oriented Systems Analysis, Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press, 1988.
25. Spivey, J. M. *The Z Notation: A Reference Manual*.
26. Toetenal, Hans, et al. "Structured Analysis - Formal Design, using Stream & Object Oriented Specification," *Software Engineering Notes*, 15(4):118-127 (September 1990). From the Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development.
27. Tse, T.H. and L. Pong. "An Examination of Requirements Specification Languages," *The Computer Journal*, 34(2):143-152 (April 1991).
28. Wasserman, Anthony I., et al. "The Object-Oriented Structured Design Notation for Software Design Representation," *IEEE Computer*, 0(0):50-63 (March 1990).
29. Yourdon, Edward. *Modern Structured Analysis*. Yourdon Press Computing Series, Englewood Cliffs, NJ: Prentice Hall, 1989.
30. Zave, Pamela. "An Insider's Evaluation of PAISley," *IEEE Transactions on Software Engineering*, 17(3):212-225 (March 1991).

### *Vita*

Captain Mary Margaret Boom, nee Zelasko, was born on 4 May 1955 in Buffalo, New York and graduated from Archbishop Carroll High School in Buffalo in 1973. Mary enlisted in the Air Force in 1973, and following basic training attended technical school for avionics maintenance (AFSC 328X4) at Keesler AFB, Mississippi. Upon completion of training, she was assigned to K. I. Sawyer AFB, Michigan where she maintained B-52Gs and KC-135As. Subsequent assignments took her to Kunsan AFB, Korea and George AFB, California to work on F-4s. In 1977, she retrained in general accounting (AFSC 672X1) at Sheppard AFB, Texas. She was transferred to Sembach AFB, Germany in 1979 and began attending classes through the University of Maryland. In 1983, Mary transferred to University of Oklahoma in Norman, Oklahoma where she completed her Bachelor of Science in Electrical Engineering (BSEE) in August of 1986. After graduation, TSgt Boom attended Officer Training School (OTS) at Lackland AFB, Texas and received her commission on 19 November 1986. She was then sent to Keesler AFB, Mississippi to attend the communications-computer course for 492X officers. After completion, she was assigned to Offutt AFB, Nebraska. While assigned there, Mary maintained interface software and provided program management and system support for the Strategic Air Command Digital Network (SACDIN). She also served as an Software Engineering Instructor for SAC programmers. In May 1991, Captain Boom entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio, in pursuit of a Master of Science degree in Computer Engineering.

Permanent address: 89 Currier Street  
Buffalo, New York 14212

### *Vita*

Captain Bradley Mallare was born on 22 October 1966 in Jamestown, New York. Upon graduation from Jamestown High School in June 1984, he received a four year Air Force ROTC scholarship to pursue an engineering degree. In May 1988, Brad graduated with honors from Clarkson University, Potsdam, New York, with a Bachelor of Science degree in Electrical and Computer Engineering. He was honored as a Distinguished Graduate (DG) from ROTC and was awarded a Regular Commission into the Air Force upon graduation. His first assignment was to the Electronics Systems Division (ESD) at Hanscom AFB, MA where he served as the Software Testing Engineer for the Command Center Processing and Display System - Replacement (CCPDS-R) Program. In May 1991, Capt Mallare entered the Air Force Institute of Technology (AFIT) at Wright-Patterson AFB, Ohio, to pursue a Master of Science degree in Computer Engineering. Upon graduation in December 1992, Mallare will be assigned to the Wright Laboratories at Wright-Patterson AFB.

Permanent address: 93 Falconer Street  
Jamestown, New York 14701